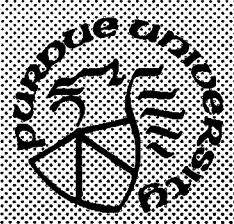
2-1-1989

DISC: A Method for Dynamic Intelligent Scheduling and Control of Reconfigurable Parallel Architectures

F. j. Weil
Purdue University

L. H. Jamieson
Purdue University

E. J. Delp
Purdue University



DISC: A Method for Dynamic Intelligent Scheduling and Control of Reconfigurable Parallel Architectures

F. J. Weil
L. H. Jamieson
E. J. Delp

TR-EE 89-14
February, 1989

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

DISC: A METHOD FOR DYNAMIC INTELLIGENT SCHEDULING AND CONTROL OF RECONFIGURABLE PARALLEL ARCHITECTURES

Francis J. Weil

Leah H. Jamieson

Edward J. Delp

School of Electrical Engineering

Purdue University

West Lafayette, Indiana 47907

Purdue University

TR-EE 89-14

February 1989

This research was supported in part by the Air Force Office of Scientific Research under Grant F49620-86-K-006.

ACKNOWLEDGMENTS

The authors would like to thank Professor Howard J. Siegel and Professor Leonard Lipshitz for their help.

This research was supported in part by the Air Force Office of Scientific Research under Grant F49620-86-K-006.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

Table Page

Appendix
Table

LIST OF FIGURES

Figure Page

Appendix
Figure

ABSTRACT

Weil, Francis J. Ph.D., Purdue University. December, 1988. DISC: A Method for Dynamic Intelligent Scheduling and Control of Reconfigurable Parallel Architectures. Major Professor: Leah H. Jamieson.

This work studies the use of intelligence-guided control of reconfigurable parallel processing systems. A reconfigurable architecture is one that can be partitioned into several independent virtual parallel machines operating in either SIMD or MIMD mode. Reconfigurable systems, while allowing great flexibility, present many scheduling and control problems. Scheduling tasks on such a system is an exponential time problem. Therefore, in an effort to achieve reduced task execution time without incurring unacceptable scheduling costs, an expert system is used to apply heuristics to approximate an optimal schedule.

When the execution time of a task is not known *a priori*, conventional scheduling methods which produce optimal or near-optimal schedules cannot be used effectively. A dynamic controller, however, is not locked into a static schedule and can reconfigure the machine and process subtasks based on the current state of the parallel processing system.

The scheduling system attempts to achieve decreased execution time by balancing the overall processing scenario of the task with the

needs of the individual routines that make up the task. Repartitioning is done when either the processor's resources need to be split among the subtasks or the processor's resources have become fragmented and need to be merged into larger partitions. The scheduler keeps track of what subtasks are potentially executable and chooses the best candidate by considering the relative importance of quickly finishing the subtask and the matching of partition data contents and subtask data needs.

CHAPTER 1
INTRODUCTION

Many of the scientific and engineering computer processing tasks that are being done today are very computationally intensive and, as such, require large amounts of time to complete. For many applications, however, it is imperative that the processing be finished quickly. One possible solution to the computational bottleneck is the use of parallel processing systems.

One of the most flexible types of parallel architectures is the reconfigurable parallel processor which can be partitioned into multiple independent virtual machines. Reconfigurable systems, however, present many scheduling and control problems. Scheduling tasks on such a system can be an exponential time problem. Therefore, in an effort to achieve reduced task execution time without incurring unacceptable scheduling costs, an intelligence-guided system is used to apply heuristics to approximate the optimal schedule.

The intelligent scheduler is part of an image understanding task execution environment layered around the parallel processor. The environment is designed to isolate the user from both the details of the underlying parallel processor and the mechanics of parallel programming. Image processing tasks are ideal for parallel processing and dynamic scheduling since they are computationally intensive, decomposable into smaller subtasks, and often have totally unpredictable execution times. When the execution time of a task is not known *a priori*, conventional scheduling methods which produce optimal or near-optimal schedules cannot be used effectively. A

dynamic controller is not locked into a static schedule and can reconfigure the machine and process subtasks based on the current state of the parallel processing system.

The research system uses a database of execution characteristics of pre-written image processing routines, rule-based heuristics, and the current system state to produce and continually update a schedule for the subtasks that comprise the overall task. The system, on input of the description of the task to be processed, produces the instructions needed by the low-level operating system to begin the task. When a subtask in a partition finishes executing, system resources become available and the scheduler directs the operating system to make any necessary processor reconfigurations and to start the execution of any indicated subtasks.

The scheduling system attempts to achieve decreased execution time by balancing the overall processing scenario of the task with the needs of the individual routines that make up the task. Repartitioning is done when either the processor's resources need to be split among the subtasks or the processor's resources have become fragmented and need to be merged into larger partitions. The scheduler keeps track of what subtasks are potentially executable and chooses the best candidate by considering the relative importance of quickly finishing the subtask and the matching of partition data contents and subtask data needs. Once a subtask is chosen for execution in a given partition, the scheduler selects the most suitable implementation of that subtask from the algorithm library. An implementation is chosen based on how well its characteristics coincide with the mode, data format, and data allocation of the selected partition and on the relative speedup for the size of the partition.

This thesis will discuss the implementation of the intelligent scheduling system and the associated information in the system database.

Chapter 2 presents an overview of parallel processing hardware. It discusses parallel processors in general and then considers

reconfigurable processors in more detail. The prototype hardware for the scheduler is the PASM parallel processing system being developed here at Purdue University. Details of the PASM system are presented at the end of chapter 2.

Inherent in the use of a parallel processing system is the need for parallel programming techniques. Chapter 3 presents several approaches to parallel processing. With this basis, an overview of the approach used by the DISC system (*D*ynamic *I*ntelligent *S*cheduling and *C*ontrol) is then presented. The types of tasks amenable to reconfigurable parallel processors are also discussed.

Image understanding tasks provide the basic processing scenarios for the prototype DISC system. Chapter 4 provides a discussion of image understanding tasks in general and why they are well suited to processing by the DISC system.

Chapter 5 presents the Image Understanding Task Execution Environment of which DISC is a part. The environment provides a system that is layered around the parallel hardware and provides the parallel processing support. Each of the components of the environment and the operation of the environment as a whole are discussed.

The DISC system is a major part of the Intelligent Operating System component of the task execution environment. Chapter 6 discusses in detail the internal working of DISC. The language choice, code layout, and implementation details are discussed. Also, the heuristics used to make DISC's scheduling and reconfiguration decisions are presented.

Chapter 7 discusses the problems associated with performance measures on a reconfigurable architecture. A survey of performance criteria at both the algorithm and task levels is presented. Examples of results from the DISC system and DISC performance measures are given.

Chapter 8 reviews the work accomplished and presents suggestions for future work.

There are three appendices to this thesis. The first appendix presents a mathematical analysis of the parallel processor reconfiguration problem. The second appendix presents a summary of the CLIPS language which was used to implement the DISC system. The last appendix gives instructions for using the DISC system and provides listings of the DISC implementation code.

CHAPTER 2
PARALLEL PROCESSORS

2.1 Overview

Parallel architectures have been of growing interest since the late 1950s [Unge58, Holl59]. During the next decade, parallel computing systems such as the SOLOMON [Slot62] and ILLIAC IV computers [Barn68, Bouk72] were proposed and studied. At that time, there was also attention given to very specific types of parallelism such as dual arithmetic logic units in a single processor [Toma67]. However, there was no firm foundation of computer hardware classification until 1966 when Flynn published his much-quoted paper [Flyn66]. In this paper, Flynn detailed the four major types of architectures: Single Instruction Stream - Single Data Stream (*SISD*), Single Instruction Stream - Multiple Data Stream (*SIMD*), Multiple Instruction Stream - Single Data Stream (*MISD*), and Multiple Instruction Stream - Multiple Data Stream (*MIMD*). These categorizations, although expanded by the advent of new architectures, still form the basis of current architecture classification.

2.1.1 Parallel Architecture Classifications

By Flynn's classification scheme, *confluent SISD* architectures are those that achieve high speed processing by overlapping the execution of the individual instructions on the sequential instruction stream

through the use of a pipelined central processor. An *SIMD* machine has multiple processors executing the same instruction on local sets of data. A much less common architecture, the *MISD*, has a single SISD execution unit that is sequentially used by a number of virtual machines. Each of the virtual machines has its own set of instructions and interaction between the processes can occur only through the data stream that is common to all machines. Finally, the *MIMD* machine has a number of self-contained processing units that each have their own instruction streams and data streams.

In addition to Flynn's classification scheme, a number of different architectures have been proposed. *MSIMD* (*M*ultiple *SIMD*) machines, such as MAP [Nutt77, Nutt77b] and the original ILLIAC IV design [Barn68], are systems which can be divided into a number smaller SIMD machines, each operating independently. *Reconfigurable* parallel architectures, which will be further discussed in the next section, are machines whose processors can be partitioned into multiple independent virtual machines each operating in either SIMD or MIMD mode. The PASM processor [Sieg81, Kueh85, Schw87] is an example of such a machine.

Some parallel processors have a large number of processing elements connected in a mesh pattern with each processor having a comparatively simple function. These computers, known as *Systolic Arrays*, are named after the heart's pumping action because of the way data is pulsed through the processors [Kung82b, Ande85]. At each machine cycle, data is moved from one processing element to the next until it emerges at the output end of the system. Variations of these computers are also known as *Wavefront Processors* for the way the leading edge of the data expands as it flows through the array [Kung82].

There has been some work done on *dataflow* architectures, in which the structure of the machine is specifically based on the flow of data primitives through the processing steps [Finn85, Kell80]. Dataflow machines are classified as such by two factors [Gajs82]. First, the

computations are performed asynchronously. An operation is performed when all required data is available. Second, there are no side effects of the operations. The complexity of the atomic operations can range from simple two-operand arithmetic operations such as addition or multiplication to large data-activated program chunks [Babb84].

Many commercial and research parallel processors, such as the CRAY series and the CYBER 205, are vector processing machines [Kasc80, Widd80]. They typically employ a high-speed pipeline architecture in which the vectors of data flow through the processing pipeline.

Many special-purpose parallel processors have also been proposed. They have hardware that is customized to process a single problem or problem domain such as image processing, speech processing, LISP programming, or artificial intelligence.

2.1.2 Parallel Architecture Components

A parallel processing system, no matter what its size, contains a fundamental set of components. The internals of a specific parallel processor, namely the PASM system, will be detailed in section 2.3. For now, only a simplified overview of generic system components will be given. Table 2.1 lists the basic components of parallel processors.

The primary components of any parallel processor are the *processing elements* (referred to as *PEs*). They are the basic building blocks of the system's computational engine and perform the specified operations on the data set. Flynn refers to these elements as *execution units.* They can be relatively uncomplicated in construction, composed of simple CPU like those in a systolic array architecture such as CESAR [Ande85], relatively complex, composed of an advanced CPU, buffered local memory, PE-to-PE network controller, external memory interface, and a control unit interface like those in some reconfigurable

Table 2.1 Basic Parallel Processor Components

Processing Elements (PEs)
Interconnection Network (ICN)
System Control Unit (SCU)
Memory System (Shared or Non-shared)
Mass Storage, I/O, Networking

processors (such as PASM) or very advanced CPUs such as those in the CRAY X/MP.

To allow the PEs to communicate among themselves (and possibly with global memory), an *interconnection network* (*ICN*) is used. Except for shared memory (which is used on some machines), the ICN is the only method available to the PEs to broadcast and receive data. The network also dictates the topology of the machine. That is, one PE can only directly communicate with another PE if those PEs are adjacent in the network. In an n-dimensional cube network (or *hypercube*), two PEs are adjacent if their addresses (their PE number in binary) differ by only 1 bit. For example, PEs 7 and 23 (00111 and 10111 in a 5-dimensional cube) are adjacent since their addresses differ by only the most significant bit. PEs 7 and 8 (00111 and 01000) are not adjacent since their binary representations differ in more than one location. Virtually any geometry is possible for a network. Some other common configurations are grid connected (4-nearest-neighbor), hexagonal and octagonal connected (6- and 8-nearest-neighbor), and various tree-like arrangements. The chosen topology is largely

influenced by the type of processing that is targeted for the machine since many classes of problems have a natural or inherent pattern of data dependencies associated with them.

Each system typically has at least one *system control unit (SCU)* whose function is to initiate, monitor, and direct the activities of the PEs. The SCU may also have other duties such as user interaction and memory control, but its main purpose is to oversee the workings of the processing system as a whole. There may, however, be several additional levels of control hierarchy between the SCU and the individual PEs. For example, the SCU might have control of two auxiliary control units, each of which has control of two more auxiliary control units, etc., until the level of the PEs is finally reached. In this case, the SCU would initiate a job on one of its auxiliary controllers and then let that controller take charge of the details of getting the job running on its subordinate PEs.

The memory systems of parallel processors are classified as *shared*, *non-shared*, or possibly some hybridization of the two. In shared memory systems, all PEs have access to a common data area and each PE can read from and write to that area. In non-shared memory systems, each PE has its own local storage area for data and can only access data from another PE through the use of the interconnection network. The type of memory system has an effect on the operation of the system as a whole since shared memory systems have synchronization and data integrity problems that are different from the synchronization and data sharing problems associated with non-shared memory systems.

Most parallel processors also have components that are not directly related to the parallel computations. As with non-parallel machines, mass storage systems such as on-line disk systems, serial and parallel I/O channels and controllers, and network systems such as Ethernet are often present.

2.1.3 Parallel Architecture Examples

There are many different types of parallel processors both on the market and under development. They range from multi-million dollar supercomputer systems to specialized architectures for image processing to plug-in processor boards for personal computers.

Examples of SIMD machines are the Connection Machine from Thinking Machines [Hill85], GF11 from IBM [Beet85], ILLIAC IV from the University of Illinois [Barn68], MPP from Goodyear Aerospace [Batc80, Batc82], and STARAN from Goodyear Aerospace [Batc74].

Examples of MIMD machines are the BBN Butterfly from BBN Laboratories [Crow85], Cm* from Carnegie-Mellon University [Swan77, Swan77b, Jone77], the Cosmic Cube from Caltech [Seit85], and the NYU Ultracomputer from New York University [Gott83].

Hybrid architectures include MSIMD machines such as MAP [Nutt77] and SPHINX [Cler87] and reconfigurable machines such as TRAC [Sejn80] and PASM [Sieg81]. There are also specialized architectures for specific problem domains such as image processing. The Pyramid Vision Machine (PVM) [Burt87] and the Semantic Network Array Processor (SNAP) [Dixi87] are two examples of such machines. Even personal computers can now enter the ranks of parallel processing. For example, Parallon Parallel Processor from Human Devices [Huma86] contains eight nodes in a MIMD configuration on a plug-in board. Up to eight of these boards can be operating simultaneously in a single system.

2.2 Reconfigurable Parallel Processors

Reconfigurable parallel architectures are the most general kinds of parallel processing systems since they allow the formation of several independent virtual machines and allow the most number of machine configurations. In this section, they will be discussed in more detail.

2.2.1 Description

A *reconfigurable parallel processor* (RPP) is classified as such by two features. First, the machine can have multiple independent virtual parallel processors existing and operating simultaneously. In some systems (e.g. PASM), each partition can be operating in either SIMD or MIMD mode. Second, the machine is capable of being dynamically reconfigured from one set of virtual machines to another. Each of the virtual machines is called a *partition* of the machine.

At this point, a description of the notation used to indicate a partitioning of a machine is necessary. A LISP-like structure will be used throughout this thesis. All PEs in a given partition of the machine will be grouped together and enclosed in parentheses. The grouping of all partitions that make up the entire machine state are then grouped with one more level of parentheses. For example, consider a repartitionable system with 8 processors numbered sequentially from 0 to 7. A partitioning with PEs 0 through 3 in one partition, 4 and 5 in another, and 6 and 7 each in their own partition would be indicated by ((0 1 2 3) (4 5) (6) (7)). The ordering of the numbers in the subsets and the ordering of the subsets themselves are not important. They will, however, generally follow the natural numbering of the underlying architecture.

To further illustrate the concept of partitioning, assume some reconfigurable processing system has a total of 4 PEs. There are 5 functionally distinct ways in which the machine can be partitioned: all 4 PEs combined into a single virtual machine; 2 virtual machines, one with 1 PE and the other with 3 PEs; 2 virtual machines, both with 2 PEs; 3 virtual machines, two with 1 PE each and one with 2 PEs; and 4 virtual machines, each with one PE. The interconnection network of the machine restricts the size and form of the possible machine partitionings. If the ICN in this example is a 2-dimensional hypercube, then there are only two ways to make two virtual machines of size two. The possibilities are ((0 1) (2 3)) and ((0 2) (1 3)). The case of ((0 3) (1

2)) is not a valid machine configuration since PEs in one partition are connected only through another partition [Sieg80, Jeng88]. That is, to get from PE 0 to PE 3, either PE 1 or PE 2 must be crossed and both are in a different partition than (0 3).

2.2.2 Applications

Reconfigurable parallel processors allow a wide range of possible applications. In its most basic configuration, that of all PEs acting as a single partition in either SIMD or MIMD mode, a reconfigurable machine can function identically to a normal SIMD or MIMD machine respectively. However, the RPP has the added ability to be configured to allow several different parts of a task to execute at the same time, or, alternately, to process several independent tasks simultaneously. Not only does this scheme allow the opportunity to minimize wasted, idle resources, but can also inherently help reduce the overall execution time of a given task. This extra speed comes from the fact that there is a nonlinear speedup as the number of PEs increases. Due to increased communications and synchronization costs, increasing the number of PEs dedicated to a job by a factor of N will almost always result in a speedup of less than N. The consequence of this fact is that it is generally more efficient to run two jobs simultaneously, each with half the available resources, than to run each sequentially, each with all the available resources.

There are many problems domains that lend themselves naturally to processing on a reconfigurable system. In addition to concurrently processing unrelated tasks, RPPs are well suited to such fields as image processing and speech processing. Any task that can be decomposed into a set of smaller, relatively independent subtasks is a good candidate for execution on a reconfigurable system. "Relatively independent" subtasks are those jobs whose only co-execution constraint is the availability of input data. Since each subtask can be

run in a separate partition of the RPP, timing constraints due to these inter-subtask data dependencies become the only major scheduling restrictions. Another advantage of using an RPP is that if a task has a decomposition that naturally lends itself to both SIMD and MIMD processing, the system can be configured as needed to allow the most advantageous use of the parallelism.

There is, however, a drawback to this amount of flexibility. A reconfigurable system can be extremely difficult to control efficiently due to the many possible scenarios and unknowns that can occur in even a relatively simple task. At best, the allocation of system resources to the subtasks of a given task is a bin packing problem [Tuom81]. Given that bin packing is an NP-complete problem [Horo78], it may not always be possible to find quickly the optimal configuration and partition assignment for a task. There has not been much work done to date on dynamic process control for reconfigurable systems. Some methods, such as the numerous variations on simulated annealing [Davi87, Moor85, vanL87], job shop scheduling [Rinn76], and genetic algorithms [Davi87], have been studied and are well suited to tasks that have performance characteristics that are known *a priori*. If a subtask to be run in a given partition has, for example, an unknown execution time, then these methods will in general not be able to produce the needed schedule and configuration for the RPP. The majority of this thesis will be devoted to exploring an intelligence-guided control scheme for tasks with both known and unknown execution characteristics.

2.3 The PASM Parallel Processing System

The target processor for this research is the PASM (*PA*rtitionable *S*IMD/*M*IMD) parallel processing system being developed at Purdue [Sieg81]. PASM is a reconfigurable parallel processing system capable of being configured into a number of independent virtual machines of

different sizes and modes. One of the design goals of PASM is to develop a system that can function as an environment for studying the use of parallelism in applications such as image processing.

2.3.1 Hardware Description

Figure 2.1 provides a diagram of the PASM system prototype. The hardware and software of PASM are each built up in levels so that the programmer need not be concerned with inappropriate details. The system can be described on three different levels: the hardware level, the interrupt level, and the reconfiguration level [Schw87].

The hardware level model consists of the physical components and connections and does not directly concern the end user. The interrupt level model contains the hardware information needed by the system programmer such as interrupt handling and memory management. This model does not contain all the physical level details of the hardware model, but still contains more information than needed by the end user.

At the highest level of abstraction, the reconfiguration level (Figure 2.2), the PASM system consists of a system control unit (which coordinates the overall function of the system), a parallel computation unit (which contains the individual processing elements and the communication network, Figure 2.3), a memory storage system (secondary storage for the parallel computation unit), a memory management system (which controls file transfers between the processors and the memory storage system), control storage (containing the programs for the micro controllers), and micro controllers (which control the activities of the PEs and are subordinate to the SCU).

In the PASM system, each of the partitions has 2^N processing elements (for some integer N), can operate in either SIMD or MIMD mode, and can switch dynamically between modes. Figure 2.3 shows the logical layout of PASM's computational engine. For the sake of

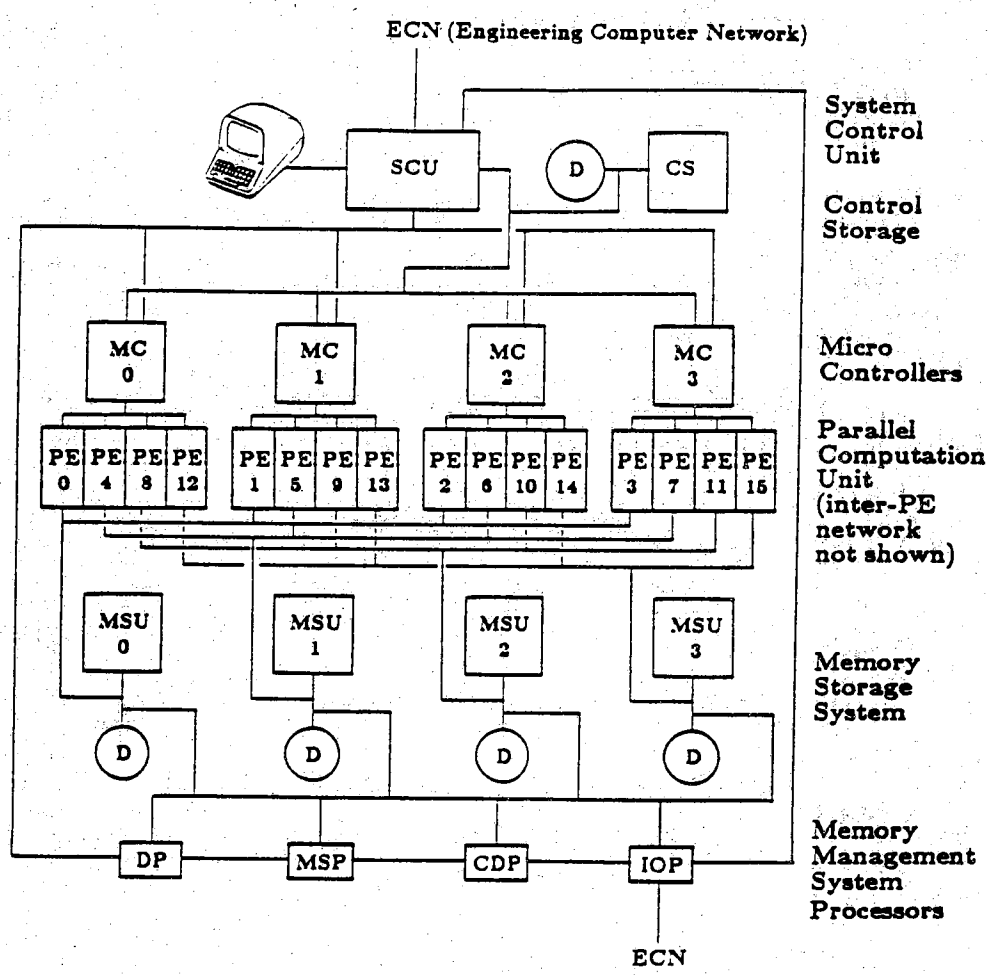


Figure 2.1 The PASM Parallel Processing System Prototype

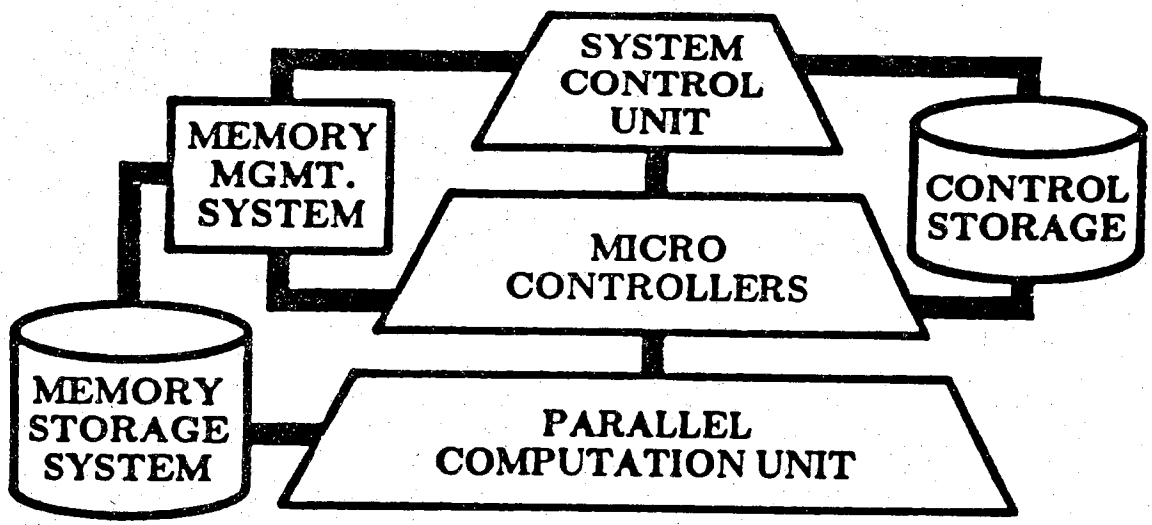


Figure 2.2 The PASM System Block Diagram

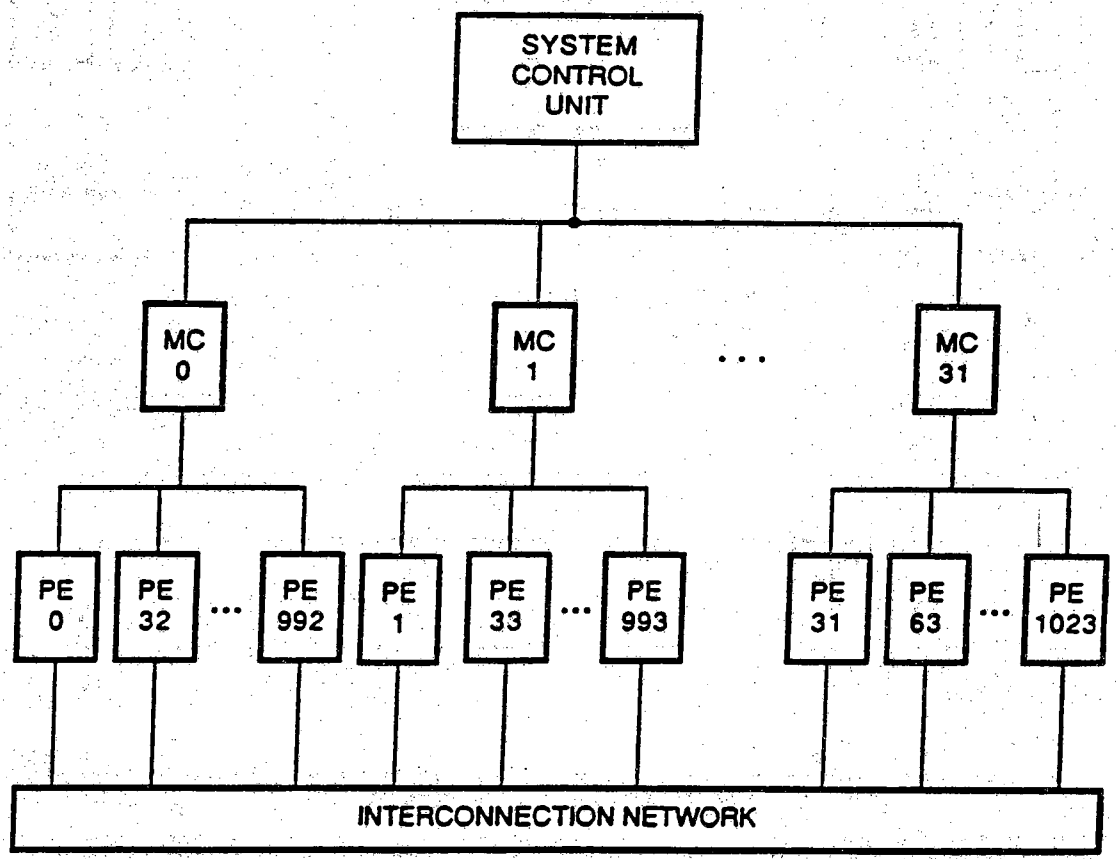


Figure 2.3 The PASM System Computational Engine

simplicity, links to mass storage, the processors executing the distributed operating system, etc. are not shown. When fully implemented, the system will have 1024 separate processing elements controlled by 32 micro controllers (MCs) and the system control unit. The SCU coordinates the activities of all other components in PASM. It performs such tasks as job scheduling and PE memory loading synchronization. Each of the MCs controls a set of 32 PEs (which is therefore the smallest partition size). The MCs, numbered sequentially from 0 to 31, are the control units for the PEs when the system is in SIMD mode and they act as overall coordinators for the PEs when the system is in MIMD mode. The MCs also provide overlapped memory loading and computations by using double-buffered memory. Each PASM processing element consists of a processor/memory pair and various communication controllers. Each PE is assigned a number which is its physical address. The interconnection network allows inter-PE communication and also helps determine the possible system partitionings. Due to the ICN hypercube topology and the overall control hierarchy, all PEs in a partition of size 2^p must have the same low-order 10—p bits in their physical address. This restriction is equivalent to stating that all MCs grouped into a partition of size P must have identical low-order 10—$\log_2$P physical address bits. It is this constraint that forces the partitions to be a power-of-two size and restricts a given MC to be in only one partition at a time.

As an example, for a particular task the system might be running with four partitions: two with 128 processing elements each in MIMD mode, one with 256 processing elements in MIMD mode, and one with 512 processing elements in SIMD mode. At a later time, the system might be configured to run 32 MIMD processes, each in a partition of 32 processing elements. If all available processing elements are not needed for the current task, the unused ones will simply remain idle.

Communication between Purdue's Engineering Computer Network (ECN) and PASM is done through the system control unit and the I/O processor in the memory management system. The user's terminal is

connected to one of the host computers which provides the development and debugging environments. Tasks are sent from ECN to the system control unit for scheduling and execution. This scheme prevents overburdening the system control unit with user I/O. The memory management system I/O processor provides access to the storage and retrieval facilities of the ECN computers.

2.3.2 Software Description

PASM utilizes several different types of software. Each controller in the system (memory management, micro controllers, etc.) executes its own code. This code is transparent to the system user since it handles data transfer, interconnection network configuring, and other tasks about which the user does not need to know the details. The system control unit contains the operating system which has overall responsibility for the functioning of the processors. Logically, one of the functions of the system control unit is to run the scheduler part of the operating system. However, there is no reason that the scheduler must physically be contained in the system control unit. It could be run on a separate processor or even in a partition of the PEs. The final location for its execution will be determined by available resources and the need to eliminate any computational bottlenecks.

The code used to implement a particular parallel algorithm is the software that affects the user the most. The programmer has to deal with the intricacies of the algorithm such as which mode to use, how the data will be divided among the PEs, and what data needs to be shared and with which other PEs. However, he or she does not have to know which physical PE is executing the instructions since the system operates in a virtual machine mode. That is, a given algorithm designed to run using N PEs will execute the same on any grouping of N PEs allowed by the interconnection network.

CHAPTER 3
PARALLEL PROGRAMMING

Using a parallel processing system requires that the user's task be broken down, or decomposed, into appropriately sized subtasks. The appropriate size is a balance of three factors:

(1) a large number of small subtasks allows a greater opportunity for parallelism than a small number of large subtasks;

(2) more subtasks require more interprocess communication and synchronization and therefore entail a greater amount of system overhead;

(3) many subtasks have an inherent or "natural" degree of parallelism and may experience a degradation of performance if too few processors (loss of speed) or too many processors (waste of system resources) are assigned to the subtask.

Once the task has been written in terms of properly sized subtasks, both the parallelized code for each individual subtask and a schedule for the subtasks to be run is needed. For most realistic tasks, neither one is trivial.

3.1 Basic Approaches

Non-parallel computers use some sort of formalized language so that the programmer can translate a conceptual understanding of a problem into concrete instructions for the computer to execute. A compiler and assembler then translate these instructions into the low-level operations of the computer. The same situations holds true for parallel computers with the added condition that the parallelism of the problem must also be made explicitly known to the computer.

There are three basic approaches to the specification of parallelism. The first method is to totally remove the burden from the programmer. Programs can be written as sequential code and the possible concurrency is detected by a compiler. The second method is to place all responsibility for parallelism on the programmer through the use of specific directives in the code. The third method is to share the burden between the compiler and the programmer. All three approaches will be detailed in the following sections.

3.2 Compiler-Detected Parallelism

Using a compiler or preprocessor to detect the potential for parallel execution in sequential code is commonly called the "dusty deck" method. The terminology is an allusion to an old, dusty deck of computer cards. The name comes from the programmer's ability to take programs written for sequential machines and to use them without modification on a parallel computer.

3.2.1 Overview

Example 3.1 lists pseudo-code that adds two two-dimensional matrices and stores the results in a third. In these and subsequent

examples, assume that **A**, **B**, and **C** have been declared as arrays with 100 rows and 200 columns for a total of 20000 elements.

Example 3.1:

```
; sequential code to add two matrices

for i from 1 to 100 do
        for j from 1 to 200 do
                C[i,j] ← A[i,j] + B[i,j]
```

Different methods of exploiting overlapped execution for code such as is given in Example 3.1 can be used depending on the target architecture. On machines that can process an entire vector simultaneously, for example, the matrices can be added by doing vector additions of the rows. Example 3.2 gives pseudo-code for this operation.

Assuming that the machine can deal with 200 element vectors in a single operation, then there is ideally a speedup of 200 from the vectorization. Another possibility would be to add the rows as vectors over all 200 columns, giving an ideal speedup of 100.

On a SIMD machine with at least 20000 processing elements, the addition could be accomplished in one step (see Example 3.3). Each PE would perform the addition and storage of a single array location from *A*, *B*, and *C*. A similar scheme could be used in MIMD processing. The speedup up in either case is 20000 (the number of array elements and PEs) minus the amount of system overhead incurred by process synchronization and control. In any case, it is up to the parallel compiler or preprocessor to detect and implement such

Example 3.2:

; vector code to add two matrices by rows

for i from 1 to 100 do
    C[i,] ← A[i,] + B[i,]

parallelism.

Example 3.3:

; code to add two matrices in parallel

C ← A + B

The principal efforts in the automatic detection of parallelism have been aimed at pipelined architectures such as the CRAY computers where the data vector is passed through a hardware pipeline for processing [Kuck80].

Some forms of programming have an inherent parallelism in their structure. In such a case, the compiler does not need to detect the parallelism, only exploit its potential. For example, logic programming has a natural ordering and concurrence built into its definition [Cone81, Cone84, vanE84, DeGr84]. Programming languages based on such

foundations are also prime candidates for specialized parallel architectures.

3.2.2 Advantages

There are many advantages to using a dusty deck approach to parallel programming. The first advantage is, as already stated, that previously written sequential code can be run directly on the parallel processor. No added time needs to be spent on modifying the code for parallelism. The savings by this method are substantial considering the volume of code that has already been written for non-parallel machines.

The next major benefit is that no special knowledge of parallel processing is needed by the programmer. Code can be written in a language and style that is familiar. New methods of programming to explicitly state the parallel execution of the code do not have to be learned. As long as there is a parallelizing compiler for the preferred language, the task of specifying parallelism falls to the computer.

Also implicit in this method is the fact that code is portable to any machine with the appropriate compiler. This portability means that not only is code directly transferable to a different machine with the same architecture, but also that the architecture itself is not important. If, for example, the user has FORTRAN code to perform a two-dimensional FFT, it will run on any machine type (SIMD, MIMD, Reconfigurable, etc.) as long as a suitable compiler exists.

The last advantage is that the code can be modified and debugged using conventional non-parallel techniques since the underlying architecture does not affect the code. Problem areas such as process synchronization and inter-PE communication do not add to the complexity of the code.

3.2.3 Disadvantages

Unfortunately, no method is without some drawbacks. The main disadvantage of dusty deck approaches is that much of the parallelism in sequential code can be hidden. As an example, consider the sequential code in Example 3.4.

Example 3.4:

```
; sequential code with ambiguous parallelism

X[i] ← X[j] + X[k]
X[l] ← X[m] + X[n]
```

The question is whether or not the two given statements can be executed in parallel. The ambiguity stems from the lack of concrete knowledge of the values of the array indices. If, for example, the subscripts have the following values:

$$i = 1, j = 2, k = 3, l = 10, m = 11, n = 12,$$

then there is no problem with overlapping the execution of the statements since there are no storage/usage conflicts. Suppose, however, that the subscripts have the following values:

$$i = 1, j = 2, k = 3, l = 2, m = 1, n = 3.$$

The outcome of executing the two statements would then depend on the order of execution. In an ambiguous case such as this, the

parallelizing compiler is forced to assume the worst possible scenario and to process the two statements sequentially.

Example 3.4 is an almost trivial example of the problems inherent in trying to automatically parallelize code. Much more complex examples of ambiguous definition and use of variables is common in sequential code. In problems where the speedup from parallel processing is critical, the lost parallelism may be too high a price to pay for the ease of use. There has, nonetheless, been a considerable amount of research done on detecting parallelism in sequential code [Alle83, Kuck74, Kuck76, Shap77] and, under certain conditions, a large percentage of the potential parallelism can be extracted [Poly86].

The other disadvantage is that the prototyping of new processing schemes can be difficult. An example task from image processing might be to recognize aircraft types from their outlines. Several different algorithms and processing schemes will have to be initially tried in order to fine tune the system for optimum results. Making such modifications can often involve extensive code changes. This disadvantage is not so much due to compiler-detected parallelism as is it to programming methods in general, but it is included here for the sake of completeness. Table 3.1 lists the advantages and disadvantages of compiler-detected parallelism.

3.3 Explicit Parallelism

Explicitly parallel code is written either when no parallelizing compiler is available for the target architecture or when minimizing the execution time of a task is desired. The programming language contains constructs to explicitly direct the parallelism of the code and it is the programmer's responsibility to make use of them. This method can be thought of as the "brute force" method in that the program has to specify what the individual PEs are doing. The programmer usually does not have to specify operations for specific PEs, though. It is up to

Table 3.1 Compiler-Detected Parallelism Summary

Advantages
• Existing code can be used without modification. • No special knowledge of parallel processing is necessary. • Code is portable. • Code is easy to modify and debug.
Disadvantages
• Parallelism in code can be hidden. • Prototyping can be difficult.

the compiler to translate the programmer's virtual machine code for generic PEs into instructions for specific PEs.

3.3.1 Overview

Parallel languages contain structures that allow the programmer to take full advantage of the parallelism in the machine's architecture. Each different type of architecture must have its own programming constructs [Hwan84]. For example, SIMD languages have some way of indicating identical parallel actions such as vector and matrix operations on local sets of data. Example 3.5 lists SIMD code for matrix multiplication [Hwan84]. The "par" directive indicates to the compiler that the subsequent loop can be executed in parallel as a

vector instruction.

Example 3.5:

```
; SIMD matrix multiplication

for i from 1 to n do {
        par for k from 1 to n do
                C[i,k] ← 0      ; vector load

        for j from 1 to n do
                par for k from 1 to n do
                        C[i,k] ← C[i,k] + A[i,j] * B[j,k]    ; vector multiply
}
```

MIMD languages contain some variation of *FORK and JOIN* or *COBEGIN and COEND* operations for parallel execution and semaphores for mutual exclusion and sharing of data among the execution streams. They employ heavy usage of synchronization primitives in the operating system. Example 3.6 lists MIMD code using COBEGIN-COEND constructs [Hwan84]. This code corresponds to the precedence graph shown in Figure 3.1.

A *precedence graph* is a method of pictorially representing the time constraints in a set of subtasks that make up a given task. The precedence graph of Figure 3.1 shows that subtask S0 must be executed first. Once S0 finishes, subtasks S1, S2, and S3 may be executed simultaneously. Once all three of these subtasks finish, subtask S4 can begin. The equal precedence of subtasks S1, S2, and S3 only indicates the possibility of concurrent execution; whether or not they are actually

```
Example 3.6:

; MIMD code for parallel execution streams

begin
        S0;
        COBEGIN S1; S2; S3; COEND
        S4;
end
```

executed simultaneously is totally irrelevant to the structure of the graph.

There have been many efforts to add parallel constructs to existing languages. Parallel programming can be done in extensions of languages such as PL/1 [Mode76], PASCAL [Brin75, Reev80], FORTRAN [Diet86], C [Diet85, Kueh85b], PROLOG [Clar84, Shap83], and LISP [Rice85]. These extensions either remove the burden of looking for parallelism from the compiler or include constructs so that code is not ambiguously parallel.

There is also a large collection of new languages that are designed for a specific architecture or are structured so that parallel analysis is facilitated. Examples of such languages include single assignment languages such as SISAL [McGr85] and BLAZE [Mehr85], zero assignment languages such as ZAPP [Slee84], MIMD languages such as CONIC [Kram84] and occam [INMO83], and various data flow languages [Broc79]. These languages cover a range of types from those in which all parallelism is explicitly stated to refined languages in which the parallel constructs are intended to aid the compiler in detecting the

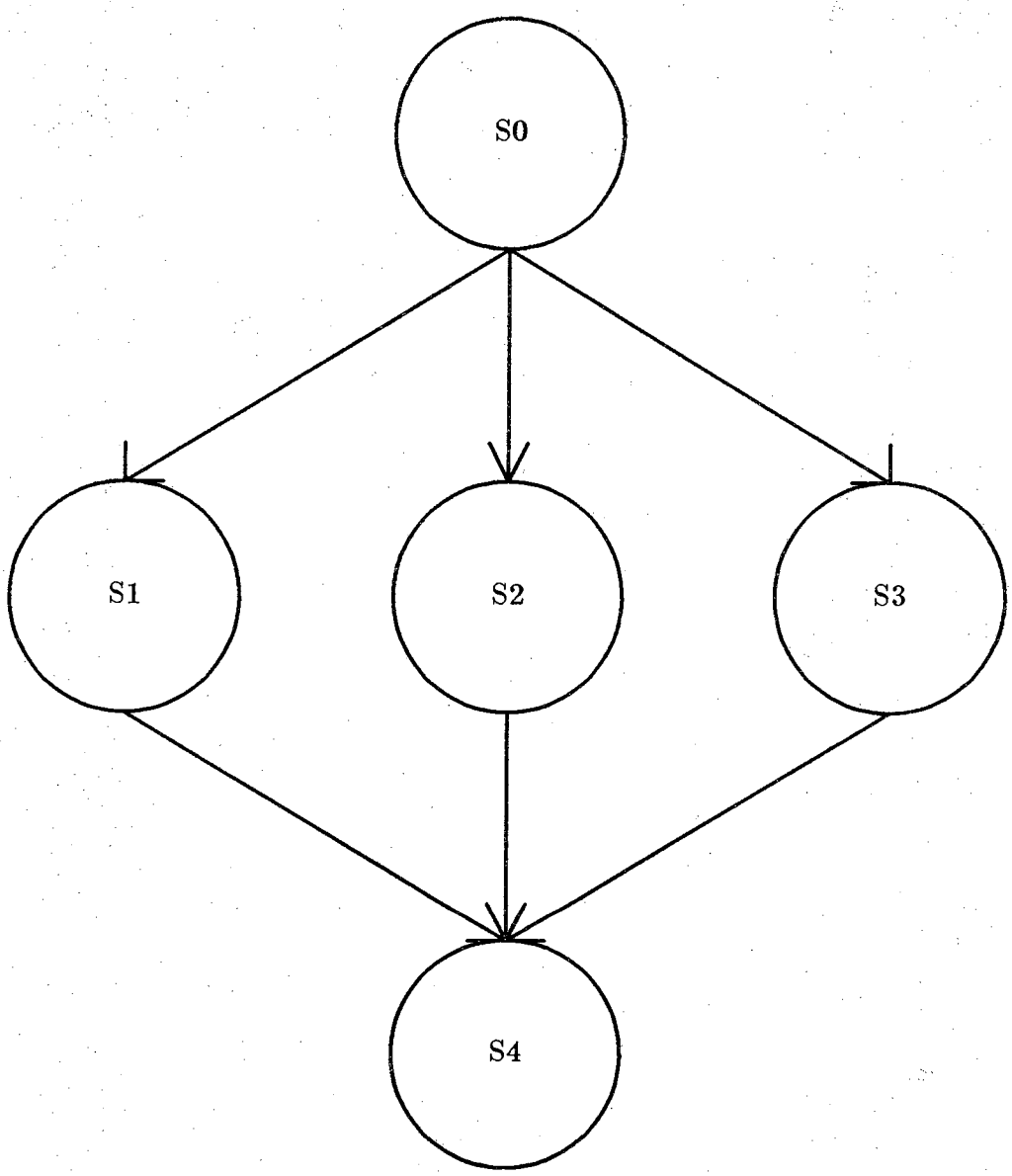


Figure 3.1 Precedence Graph for Example 3.6

parallelism.

3.3.2 Advantages

The advantage of explicitly parallel code is the potential for very fast and efficient code. Since the programmer must specify the parallelism, the problem of hidden parallelism due to ambiguous code is removed. Also, since the programmer is targeting a specific architecture, the code can be tailored to take advantage of the parallelism and the PE interconnections inherent in the machine. In this manner, wasted system resources due to code structure inappropriate to the machine type can be avoided.

Since the programmer has control of the parallelism, code segments with critical speed restraints can be tuned for optimal performance. In a parallelizing compiler, such optimization is not possible since the final execution of the code is out of the control of the user.

3.3.3 Disadvantages

User specified parallelism has several disadvantages in terms of programmer productivity and algorithm development. The programmer must have a good understanding of both the parallel constructs in the language and the underlying architecture. This understanding can entail not only learning a new language, but also learning a new style of programming. Thinking about temporal precedence in code is not something generally dealt with in sequential programming.

Another drawback of parallel code is non-portability. Different machines can have totally different languages even if the architectures of the machines are similar. Code also has to be written for a specific

mode (SIMD or MIMD) of parallelism. Once an algorithm is coded this way, it is tied to that specific architecture. With a language such as FORTRAN, code can be ported to any machine supporting the language. Explicitly parallel languages can only be compiled for a given machine if the underlying architecture supports the view of parallelism inherent in the language. On danger of this lack of portability is that the architecture imposes a certain view of parallelism on the algorithm. Instead of developing a parallel algorithm and then implementing it in the chosen language, the algorithm development has to be specifically tailored to the parallelism of the machine. Incompatibilities between the natural parallelism of an algorithm and the parallelism of the underlying architecture can force very artificial methods of programming.

Debugging parallel code can border on the impossible. Since the operation of the code depends on the specified parallelism as well as the implementation of the algorithm, a program's validity cannot be readily verified. If an error occurs in processing, the bug could be in process synchronization, inter-PE communication, or incorrect precedence as well as in the implementation of the algorithm. Debugging techniques have to employ methods of checking both the intraprocess and interprocess activity. Work on debuggers for parallel systems is still in the very early stages of development [Houg87].

Explicitly parallel code, as with compiler-detected parallelism, has an inherent difficulty with system prototyping. Finding a suitable implementation of a task can involve extensive code changes as well as changes in the indicated parallelism of the code. Table 3.2 summarizes the advantages and disadvantages of explicitly parallel programming languages.

Table 3.2 User-Directed Parallelism Summary

Advantages
• Optimally fast code is possible. • Efficiently utilized hardware is possible.
Disadvantages
• Programmer must understand parallel processing. • Code is not portable. • A certain view of parallelism is enforced. • Debugging is difficult. • Prototyping can be difficult.

3.4 Hybrid Methods

Given the problems associated with the dusty deck and brute force methods of parallel programming, efforts have been made to use the advantages of both methods while avoiding their drawbacks. One such system is the parallel software environment PARSE which is currently being designed for reconfigurable, non-shared memory parallel architectures [Casa87].

PARSE is an integrated collection of language interfaces, debugging tools, and analysis tools which allow the user to select the most suitable form for expressing the parallelism of a problem. A given problem is expressible within PARSE at a number of levels. The user may represent the problem as a knowledge-based logic program

(KBLP), sequential or parallel algorithms, communicating finite automata (CFA) [Aho79, Bran83], C code, Refined C code (RC) [Diet85], explicitly parallel C code (XPC), or the actual object code for the target machine.

The tools available in PARSE allow the translation of the problem in the user's chosen abstraction level to the code that is executed in the PEs (the object code). These tools include compilers for the KBLP, RC, and XPC code; preprocessing translators for C code and the CFA model, and analysis tools that process and help improve RC code and CFA model code.

While this system does address the problems of hidden parallelism in sequential code and non-portability of parallel code, it does not completely solve all the problems associated with parallel processing. The prototyping of tasks is still inherently difficult although the high-level modeling tools available in KBLP and CFA offer a reduction in the problem. Also, PARSE reduces but does not totally alleviate the problems of hidden parallelism and difficult debugging that are associated with compiler-detected parallelism and explicitly parallel code.

The DISC system presented in this work takes a different approach to solving problems associated with parallelism and will be discussed in the next section. DISC focuses largely on facilitating the prototyping of tasks and scheduling tasks whose algorithms have unknown execution times.

3.5 The DISC Method

The DISC (Dynamic Intelligent Scheduling and Control) system processes tasks on reconfigurable architectures. It addresses the problems of parallel processing from the standpoint of trying to allow effective use of a parallel processing system by an end user who has no parallel programming experience [Weil87, Weil88]. Instead of forcing

such a user to employ a parallelizing compiler, the basic building blocks necessary to use the parallel processor effectively are provided.

3.5.1 Overview

The DISC system will be discussed in some detail in chapters 5 and 6. For now only an brief overview will be given. In the DISC system, the user specifies the task as a sequential listing of the subtasks (or algorithms) necessary to complete the task. The primitives of the language are the actual algorithms. Example 3.7 shows a listing of DISC code for an image processing task. This code locates tanks in forward-looking infrared radar images. Two things are immediately apparent about this code. First, the program is comprised of a sequential listing of the appropriate image processing routines such as would be seen in a language such as C. Second, there is no direct indication of how a subtask such as Median_filter is to be implemented.

At the heart of DISC is a library of prewritten routines that have been optimized for the target architecture. Table 3.3 lists examples of routines that will be included in an image processing library. The user's task is broken down into subtasks that are present as algorithms in the system library. Conditional branching is provided through the *while* construct which has a syntax similar to that of the *while* loop in C.

DISC utilizes database information about the library routines and the user's task specification to assemble the required parallel code and to generate an initial schedule and configuration for the task. DISC then monitors the state of the parallel processing system and, when subtasks in individual partitions finish thereby freeing system resources, directs the parallel processor to perform any necessary reconfigurations and to begin the execution of indicated subtasks. This cycle continues until the entire task has been completed.

Example 3.7:

```
; DISC code for an image processing task

Median_filter (INPUT-IMAGE, A)
Scene_model ("TANKS - FLIR DATA", B)
Edge_detect (A, C)
Edge_link (B, C)
while (Edge_continuity (C) < 0.9) {
        Edge_link (B, C)
}
Texture_analysis (A, D)
Boundary_trace (B, C, D, E)
Shape_analysis (E, F)
Region_formation (B, D, E, G)
Object_recognition (F, G)
```

3.5.2 Advantages

DISC allows a user with no parallel processing knowledge to take advantage of the underlying architecture of the target system. There are no parallel constructs in the language and the listing of subtasks in DISC code closely follows the familiar style of languages such as FORTRAN and C. Also, a user's task, once specified in DISC, is directly portable to any machine running the DISC system.

Code that is fast and makes efficient use of the abilities of the underlying architecture is possible for each routine in the system library. For each algorithm used in the task, the appropriate code in

Table 3.3 Example Library Algorithms

$\nabla^2 G$	Hilbert Transform
Algebraic Reconstruction Algorithms	Histograms
Blackboard-based Segmentation	Homomorphic Filtering
Block Truncation Coding	Hough Transform
Canny Edge Detector [Cann86]	Huffman Coding
Constrained Deconvolution	Huffman Shift Coding
Contour Tracing	Inverse Filtering
Correlation	Kalman Filtering
Cosine Transform	Karhunen-Loeve Transform
Covariance	Laplace Filtering
Differentiation	Maximum Likelihood Classification
Edge Linking	Medial Axis Transform
Edge Thinning	Median Filtering
Filtered-backprojection Reconstruction	Min-max Filtering
Fourier Shape Descriptors	Predictive Compression
Fourier Transform	Rank-order Filtering
Gaussian Smoothing	Region Growing
Gray Level Correction	Region Linking
Gray Scale Modification	Rule-based Object Recognition
Hadamard Transform	Run-length Coding
Hankel Transform	Sobel Operator
High-emphasis Filtering	Thresholding

the library will have been optimized for the architecture and mode of operation of the machine.

DISC facilitates the rapid prototyping of user tasks. Since changing the processing used for a particular task only entails modifying the list of algorithms, many different processing schemes can rapidly be tested so that the best strategy can quickly be determined.

Many algorithms have nondeterministic execution times and therefore cannot be scheduled by static methods. As stated in the previous chapter, strategies that try to schedule the task as a whole (simulated annealing, job shop scheduling, etc.) cannot be used unless all subtasks that make up a given task have execution characteristics that are known *a priori*. The dynamic scheduling used in DISC is not affected by unknown execution times.

3.5.3 Disadvantages

The main disadvantage of the DISC system is that the type of processing that can be done is limited by the resident library. The way the system is currently configured, a user can only specify algorithms that previously have been entered into the library. However, the processing environment of which DISC is a part contains methods for dealing with this situation and will be detailed in chapter 5. Since the library exists at the algorithm level, only tasks that can be decomposed into relatively independent subtasks are candidates for processing by DISC.

Execution of the tasks is not necessarily optimal even if the code segments that makes up the subtasks are. Since the entire task is not processed as a whole due to the dynamic scheduling scheme, inefficiencies in subtask interaction can result. There is also some execution-time overhead incurred by the DISC system itself during runtime, but this overhead is relatively small and should not affect the overall processing time to any large degree. Table 3.4 lists the

advantages and disadvantages of the DISC system.

Table 3.4 DISC System Summary

Advantages
• No user parallel programming knowledge is required. • Code is portable. • Fast and efficient code is possible. • Rapid prototyping is promoted. • Algorithms with unknown characteristics can be scheduled.
Disadvantages
• Currently limited to using library algorithms. • Overall execution is not necessarily optimal. • Some overhead is incurred by the DISC system.

3.5.4 Allowable Processing Types

In order for a task to be processable by DISC, it must be able to be broken down into a series of subtasks. For many problem domains, this type of top-down decomposition follows the natural structure of the problem. Examples of such problem domains include image processing, speech processing, computer graphics, and computer animation. Virtually any task that is of a sufficiently high level can be processed in this manner. For example, while DISC may not be

appropriate for studying different methods of matrix inversion, linear algebra algorithms such as matrix inversion could certainly be in the library as primitives for use in other tasks.

The DISC system uses information about the characteristics of the library algorithms to make its scheduling and reconfiguration decisions. The next section discusses several different parameters by which algorithms may be classified. DISC uses a subset of these parameters to obtain the information necessary to make its decisions.

3.6 Characteristics of Parallel Algorithms

The characteristics of parallel algorithms can be split into two categories. The first class is the physical characteristics of the algorithm such as the type of parallelism used in the algorithm (SIMD, MIMD, etc.). These characteristics state the details of the implementation of the algorithm. The second class is the execution characteristics of the algorithm such as the expected execution time. These characteristics indicate how the implementation will behave during execution. The distinction between these two classes is not always clear cut since they often have a direct or indirect effect on each other. For example, the execution time of an algorithm is dependent on the mode of parallelism since different modes require different programming techniques. The distinction is solely for discussion purposes.

The advantage of classifying an algorithm with a set of characteristics is that it allows diverse parallel algorithms such as FFTs [Jami86], block truncation coding [Mudg82], histograms [Sieg81b], and normalized Fourier descriptors [Smit84] to be compared on a common basis.

3.6.1 Physical Characteristics

There are a number of physical algorithm characteristics that can be used [Jami87]. Table 3.5 lists some of these characteristics.

Table 3.5 Physical Algorithm Characteristics

Algorithm mode
Data allocation
Data dependencies
Data granularity
Data types and precision
Degree of parallelism
Fundamental operations
I/O requirements
Module granularity
Uniformity of operations

The algorithm mode specifies the processing mode of the algorithm (SIMD, MIMD, pipelined, etc.). The data allocation states how the data is distributed among the PEs (by row, column, region, etc.). The data dependencies dictate the inter-PE communication requirements and the data allocation among PEs. Data granularity is the size of the smallest amount of data being processed as a unit. Data types and precision are the characteristics of the atomic data units. The degree of parallelism of an algorithm is the amount of processing occurring simultaneously. The fundamental operations are the types of processing being done be the PEs (arithmetic, etc.). The I/O

requirements specify the input and output data requirements of the algorithm. The module granularity is the amount of processing in the algorithm that can be done independently of other processing. The uniformity of operations specify the similarity of the processing done on a given set of data.

3.6.2 Execution Characteristics

As with physical characteristics, there are many parameters by which the execution characteristics of an algorithm may be classified [Jami87, Sieg82]. Table 3.6 lists some of the possible execution characteristics.

The efficiency of an algorithm is a measure of the amount that a single PE contributes to the decreased processing time. The execution time is the amount of time spent processing the algorithm. This characteristic often cannot be predicted *a priori*. The inter-PE communication specifies the load placed on the interconnection network. The memory requirement is the amount of local data storage needed by the algorithm during runtime. The number of PEs required states the maximum number of PEs that the algorithm will need during runtime. Overhead ratio is the ratio of the overhead involved in processing an algorithm to the total execution time for that algorithm. Process synchronization needs are the time constraints placed on the execution of different parts of the algorithm due to data precedence and global data access rights. Processor utilization measures the amount of time the PEs are actively involved in processing the algorithm. Redundancy gives a measure of the amount of excess operations that are performed by the PEs during processing. The speed is the amount of atomic data elements that are processed during one unit of time. Algorithm speedup is the ratio of the execution time of an algorithm for N PEs to the time for the algorithm on a single PE. Finally, a static algorithm is one that does not generate new processes during

Table 3.6 Algorithm Execution Characteristics

Efficiency
Execution time
Inter-PE communication
Memory requirement
Number of PEs required
Overhead ratio
Process synchronization needs
Processor utilization
Redundancy
Speed
Speedup
Static or dynamic nature of algorithm

execution while a dynamic algorithm will spawn at least one new process.

The set of parameters that the DISC system uses are detailed in chapter 5. Since DISC categorizes algorithms by their classification parameters, there must be a set of prewritten algorithms in the library. Although these algorithms could potentially be from any area of computer processing (speech processing, signal processing, computer graphics, etc.), image understanding was chosen as the paradigm for the research into and development of DISC. Chapter 4 will discuss image understanding tasks in some detail and show why they are ideal for the type of dynamic scheduling used by DISC.

CHAPTER 4
IMAGE UNDERSTANDING TASKS

Image understanding (also known as computer vision) has been chosen as the prototypical processing area for the DISC system. Due to the characteristics of these tasks, they are ideal for implementation on reconfigurable parallel processors. The following sections will overview image understanding tasks and discuss why they are suitable for DISC processing.

4.1 Overview

Credit for designing and implementing the first computer vision system is usually given to L. G. Roberts [Robe65]. With the advent of computer technology and the desire to make intelligent machines, the application of computers to vision seemed a natural step. Two major facts soon became apparent. The first was that computers could easily perform vision tasks that humans found difficult or impossible (determining overall brightness, specific color value, etc.). The second fact was that many tasks that a human could do with little or no thought (or effort) where extremely complex on a computer (edge detection, stereo correlation, etc.).

Unfortunately, most vision systems require the tasks that a human would feel were trivial. Tasks such as autonomous navigation, target recognition, and depth perception through stereo correlation are an

integral part of many practical vision systems.

4.1.1 Terminology

At this point, a few distinctions between terms need to be made. There are two general terms used to denote imaging procedures: image processing (also known as picture processing) and image understanding. Image understanding is a part of image processing that falls under the category of high-level processing [see Figure 4.1].

Low-level (or early processing) routines are those that perform straightforward and relatively simple functions. They are generally of a "pixels in, pixels out" nature. Examples of low-level routines would be Laplacian filtering, the Prewitt operator [Prew70], Robert's cross [Robe65], the Hueckel operator [Huec71, Huec73], gray value thresholding, and Gaussian smoothing.

Mid-level routines tend to deal more with structures in the image than with individual pixels. These routines are generally more complicated than low-level routines and include such operations as region segmentation and edge linking.

High-level routines are those that try to build a consistent interpretation of the input image. The knowledge contained in the internal representation of the scene can then be presented in the appropriate form. For example, the system might output a natural language description of the scene, track an object in a series of images, or provide navigational information. Image understanding programs are often written as expert systems since there is no general algorithm for the consistent labeling of segments in an image. However, it is possible to write high-level routines that contain no native intelligence. For example, a description of a scene could be obtained using a statistical pattern recognition approach.

There are no rigid rules used to make the distinction in processing levels. A program of one level is usually assumed to include routines

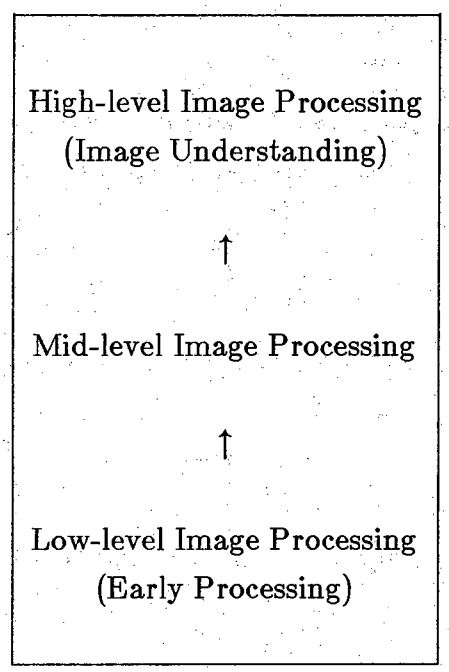


Figure 4.1 Digital Image Processing Hierarchy

from the lower levels. That is, a high-level program uses as input an image that was processed by both mid- and low-level routines. These classifications, however, are largely cosmetic and serve only as a common basis for discussions.

An image processing job of any level of complexity is called a *task*. It is the overall processing that is desired for the given application. An example image understanding task might be to find the tanks in infrared data. It does not specify exactly how the recognition is to be accomplished, only that that is what is desired. The task is broken up into *subtasks* which state what processing needs to be done to complete the task. That is, the subtasks form a top-down decomposition of the

given task. Obviously, there can be subtasks within subtasks and the labeling of a procedure as a task or subtask depends only on the current processing context. At the lowest level, the subtasks are the actual algorithms for the processing. All higher levels of subtasks exist only as abstractions of the description of the overall plan for the processing.

4.1.2 General Image Characteristics

A digitized image consists of a two-dimensional array of intensity values. The image, therefore, has been quantized in two ways. First, the spatial resolution is determined by the number of sample points per unit area that the input device has and dictates the amount of detail that is retained by the digitized image. An image size of 256 by 256 pixels is considered low resolution while high resolution images can be several million pixels on a side. Second, the size of the word used to store each pixel value determines how many different brightness values (or gray levels) can be distinguished. Quantizing an image to N bits gives 2^N gray levels. Also, the red, green, and blue components of an image are often quantized and stored separately. As is apparent from the above data, one image can present a huge amount of data for storage and processing. A 256 by 256 image quantized to 8 bits occupies 65,536 bytes (64 Kbytes) while a 2048 by 2048 three-color image quantized to 16 bits occupies 25,165,824 bytes (24 Mbytes).

Digitized images can be classified in one of three categories: binary, black and white, or color. Binary images are those that have intensity quantized to a single bit (either black or white). Black and white images are those that have intensity quantized to an arbitrary number of bits but preserve no color information. Binary images could be considered a subset of black and white images with one bit of quantization but are considered separately because of the large differences in processing techniques used to analyze and manipulate

them. Color images consist of separate images for each color (usually red, green, and blue) which are combined to yield the composite picture.

Binary images take the least amount of time to manipulate since many of the operations can be simplified as a result of the one bit quantization. Multiplications, additions, and comparisons are reduced to simple Boolean operations. However, this simplification is at the expense of the information contained in the intensity values such as distinct region boundaries. Color images are effectively a set of three black and white images which are processed independently and combined (added) on a color display device. Color images convey the most information since two different colors with the same gray scale value are indistinguishable in a black and white image. The choice of the resolution and the use of color information is dependent on the current processing application.

4.2 Image Processing and Parallelism

Many image processing tasks readily lend themselves to parallel processing. That is, the structure of the image itself combined with the type of processing desired suggest a natural parallel programming approach. These same qualities make the tasks ideal for processing by the DISC system.

Image understanding tasks are composed of a mixture of algorithms from all three levels of the processing hierarchy. The tasks, when decomposed into their component subtasks, consist largely of well-known algorithms. Whether the task is identifying aircraft from their outlines or navigating an autonomous land vehicle, the core of the processing is based on fairly generic routines such as texture analysis, edge detection, smoothing, region formation, etc..

Image understanding tasks have two properties that make scheduling by classical techniques difficult or impossible. First, mid-

and high-level image processing tasks often have an unpredictable execution time. As stated previously, most static scheduling techniques are not designed to handle this case. Second, the exact processing scenario is often unknown. The final sequence of algorithms used to process the task depends on properties of the input data. For example, many different types of smoothing and filtering can be applied depending on the amount of noise in the input image. Static scheduling techniques cannot be used when the processing to be done is not finalized before runtime.

Image processing is ideal for use in a parallel processing system since most realistic tasks tend to have very high execution times. The low speed is due to both the computational complexity of the individual steps in the processing and the huge amounts of data that have to be processed. The inherent parallelism in the algorithms and the images make for a natural mapping onto parallel processors. Image understanding tasks often have to be executed quickly due to the nature of the task. Recognizing hostile aircraft in a tactical situation would be useless unless it could be done in near real time.

There have been a large number of diverse image processing routines written for parallel processors [e.g. Mudg82, Rice85b, Tsao82, Walt78, Warp82]. Even algorithms that do not suggest a natural degree of parallelism can be structured so that parallel code can be generated. The DISC system therefore has a large amount of code available for both SIMD and MIMD modes of operation. Example 3.7 of the previous chapter shows an example image understanding task that includes processing steps from all three levels of complexity, has an unknown execution time, and has a non-deterministic processing scenario due to the edge continuity constraint.

CHAPTER 5
THE IMAGE UNDERSTANDING TASK EXECUTION ENVIRONMENT

In an effort to supply an intelligent, interactive environment for the processing of image understanding tasks, an *Image Understanding Task Execution Environment* is being created. The various layers and components of this system are detailed in the following sections.

5.1 Overview

The overall system model for executing an image understanding task is shown in Figure 5.1 [Delp85], illustrating the interactions among the various components of the environment. An alternative view of this model is shown in Figure 5.2 [Schw87], where the knowledge bases and the algorithm databases for each part of the system are grouped according to their levels of operation. Each of the components of the system will be discussed in detail in subsequent sections.

There are several types of control and data flow shown in Figure 5.1. The heavy double arrows represent the flow of the user's task from the conceptual level at the task originator to the code level at the parallel processor. It should be noted that although the term "user's task" is used, it is perfectly feasible for the task originator to be non-human such as would be the case with an autonomous land vehicle. As the task moves toward the parallel processor, the various processing

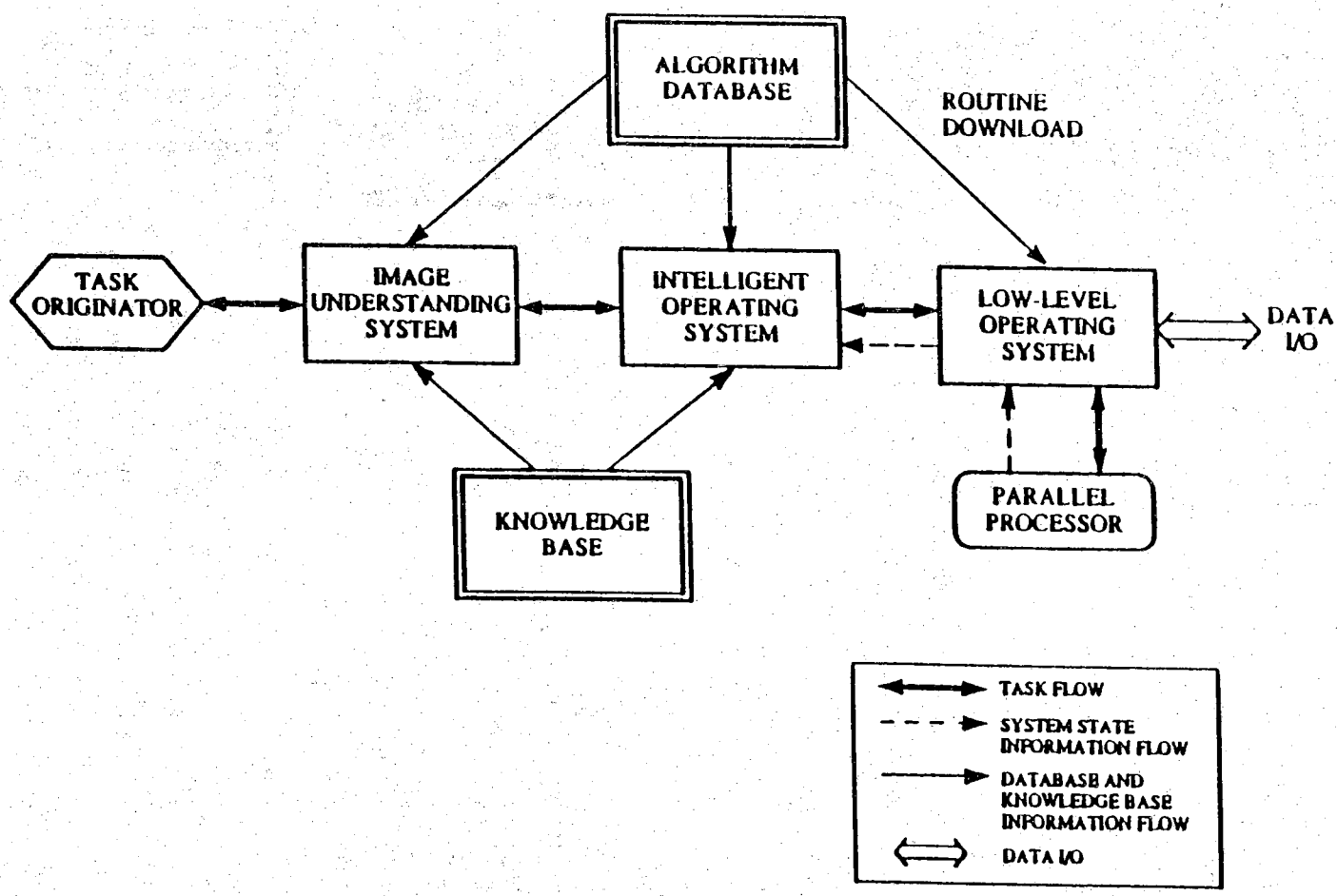


Figure 5.1 The Image Understanding Environment Overview

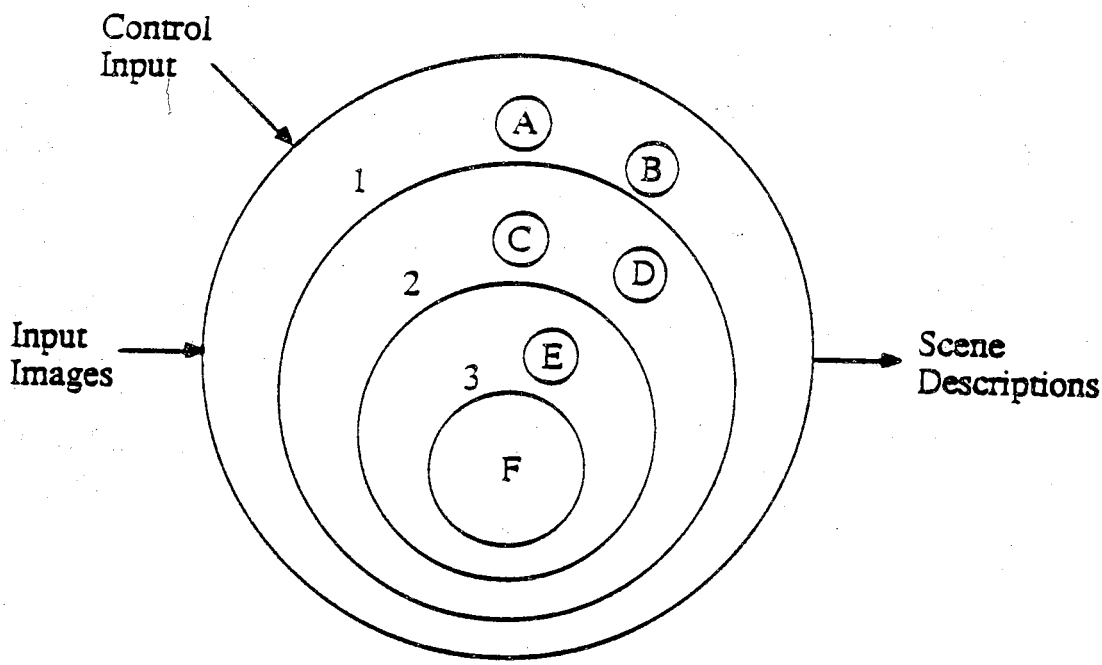


Figure 5.2 Alternate View of the Image Understanding Environment

stages analyze and manipulate the task specification so that the appropriate code can be run.

The dashed-line arrows represent the flow of information about the current state of the parallel processing system. This information includes such data as the system configuration, which partitions are idle and which are in use, and intermediate processing results. The information is used by the Intelligent Operating System in making scheduling and configuration decisions.

The thin solid arrows represent the information contained in the database and knowledge base that is used by the processing components of the environment. Note that the Algorithm Database is connected to each of the three processing components (the Image Understanding System, the Intelligent Operating System, and the Low-Level Operating System). This common thread helps allow a natural task flow from the task originator to the parallel processor. The database and knowledge base will be further discussed in later sections.

The final type of flow is the data I/O and is represented by the double hollow arrow. Loading and storing image data and results to and from the parallel processor is handled by the low-level operating system (OS).

The main research for this thesis encompasses the Intelligent Operating System, Algorithm Database, and Knowledge Base components of the environment (see Figure 5.1). The Intelligent Operating System and the Knowledge Base together make up the core of the DISC system. As an integral part of the research, the algorithm characteristics contained in the Algorithm Database were developed and applied.

5.2 The Image Understanding System

5.2.1 Overview

The Image Understanding System (**IUS**) acts as an interface between the task originator and the Intelligent Operating System. It contains information about which algorithms are used to perform a given subtask in the user's task specification. Each subtask may be performed by more than one algorithm, where each algorithm has different image analysis performance characteristics which are stored with the algorithm in the Algorithm Database. The execution order of the subtasks may be represented as a data dependency graph (**DDG**), indicating which subtasks can be done simultaneously and which must be done sequentially with respect to the other subtasks. The exact processing indicated in the data dependency graph may vary during task execution based on intermediate results that are derived. This data dependency graph is stored and maintained by the Image Understanding System.

5.2.2 Algorithm Prototyping

As was stated in chapter 3, the main disadvantage of the DISC method is the requirement of a library of prewritten algorithms. This restriction is partly reduced by the Image Understanding System.

Before testing a new algorithm, the code for the parallel implementations of the algorithm (Circle E in Figure 5.2) and the information about those implementations (Circle D in Figure 5.2) must be supplied. One issue that arises is determining when and to what extent in the prototyping process should the IUS tools assist the user in providing the information to the Algorithm Database.

Developing the parallel implementation code is not easy. The current state in the development of parallelizing compilers does not allow programmers to be as removed from the hardware as in the case of serial computers. For an entirely new algorithm, the parallel code in the Algorithm Database is either explicitly parallel code or is generated by a parallelizing compiler.

An alternative to generating the code and determining the parallel characteristics every time an algorithm is entered into the Algorithm Database is to exploit the knowledge the system possesses about the parallel implementations of other algorithms. This knowledge is in the form of the code and information about each algorithm stored in the Algorithm Database. An operation based on expert system concepts, referred to as *cloning* [Chu87], allows a starting point for algorithm development. Through an interactive interface, the cloning process would ask the user to note which algorithm in the current database most closely resembles the algorithm that is to be added. The user would be prompted for further information required by the system about the algorithm. By using features that characterize parallel algorithms, the steps that the cloning process should take to make the necessary changes can be stated explicitly in rules. Hence, the cloning process can build a new entry in the Algorithm Database based on modifying existing ones. Approaches to implement this process are currently under study.

5.2.3 Data Dependency Graph

Figure 5.3 shows the data dependency graph for the task given in Example 3.7. The structure of the graph is different from that of a *precedence graph* in that time dependency information is represented only as input data requirements. That is, the arcs represent data flow as opposed to control flow information. The DDG also does not conform to the exact definition of a *data flow graph* [Davi82, Hwan84].

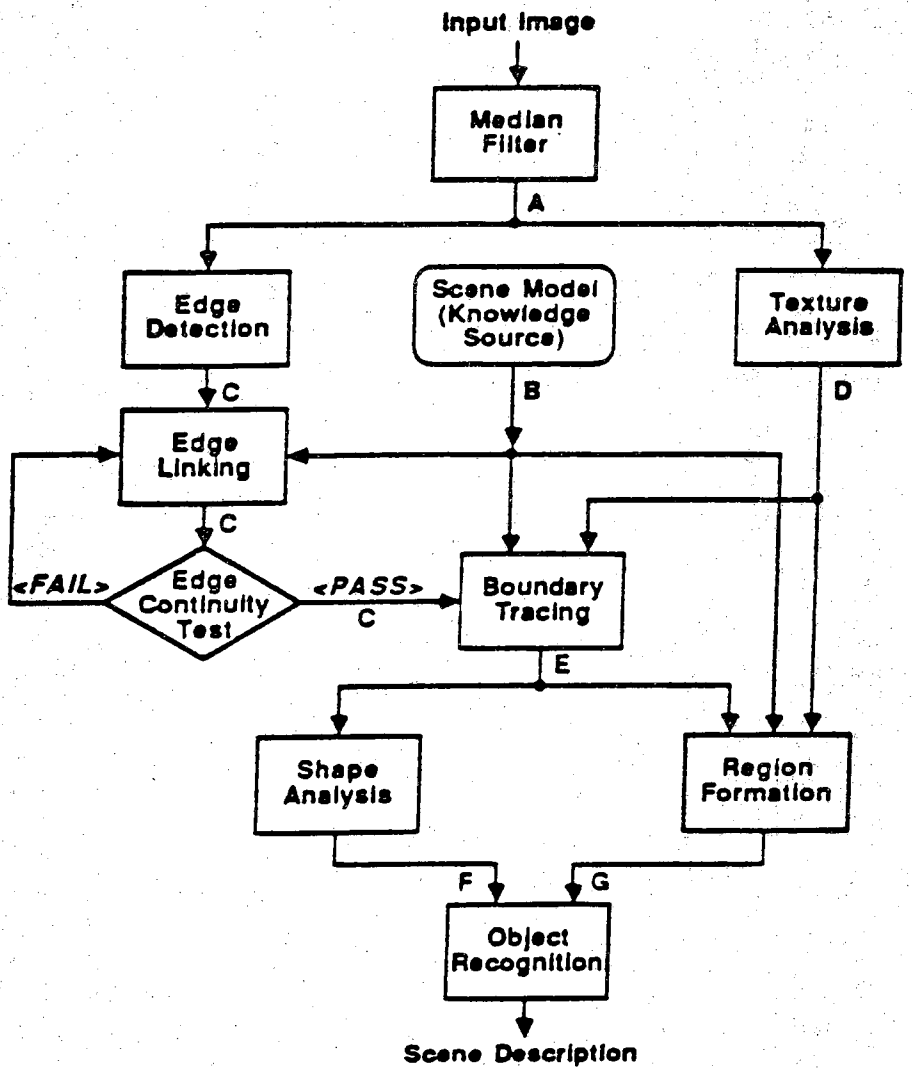


Figure 5.3 Data Dependency Graph for an Image Understanding Task

Any looping constructs in the task specification can be translated directly to a *test-and-branch* structure such as the Edge Continuity Test node in Figure 5.3. The information contained in the DDG is therefore sufficient for the scheduling of the task.

Once the Image Understanding System has built the DDG from the task specification, possibly through interaction with the user, this information is passed on to the Intelligent Operating System for further processing.

5.3 The Intelligent Operating System

5.3.1 Overview

The Intelligent Operating System (*IOS*) acts as an interface between the IUS and the parallel processor's low-level operating system. Its function is to take to user's task description in the form of a data dependency graph and produce the scheduling and control information necessary to complete the processing of the task.

The IOS has three major components. First, there is an interface to the IUS which does preliminary processing on the DDG. Second, the main component of the IOS is the scheduling and control expert system (DISC) whose function is to direct the low-level OS on appropriate algorithm execution and system reconfigurations. The internal working of the DISC system is the subject of chapter 6. The final component is the interface to the parallel processor's low-level operating system. It is through this mechanism that DISC provides the low-level OS with the necessary instructions.

All three components interact heavily with each other and there are no clear divisions between them in the DISC code. The distinction made here is purely functional and for discussion purposes only.

5.3.2 Image Understanding System Interface

There are three preprocessing steps that are performed when a DDG is received from the IUS. The first step is to translate the DDG into a form that is more suitable for processing. The next step is to read the modified DDG into the system and store the information in an appropriate internal form. The final step is to load any needed database information into the system.

The DDG contains several types of information. It lists the processing that is to be done in the task, what inputs are used and what outputs are produced by each algorithm, and it provides algorithm precedence information. The algorithms selected and their inputs and outputs are static information and can be stored directly. The execution constraints implied by the data requirements have to be extracted from the graph.

A DDG generally contains redundant precedence information. This redundancy can only complicate processing and is therefore removed from the graph. The resultant non-redundant graph is termed a Reduced Data Dependency Graph (*RDDG*). The redundancy comes from sequences of data dependencies. For example, suppose there are three algorithms in a task (X, Y, and Z) with a DDG as shown in Figure 5.4. Algorithm Y depends on data from algorithm X and algorithm Z depends on data from algorithms X and Y. In terms of precedence information, the arc labeled "b" in the DDG is unnecessary. Since Z depends on data from Y and Y depends on data from X, there is already an implied data dependency from X to Z. During the translation of the DDG to the RDDG, all unnecessary arcs are removed. The RDDG then contains the minimum amount of information required to maintain the precedence information present in the original DDG. Figure 5.5 shows the RDDG for the DDG of Figure 5.4.

The graph reduction is done by DISC when the task is initially read by the IOS. The RDDG is constructed by building descendant lists by applying the following rule to each node:

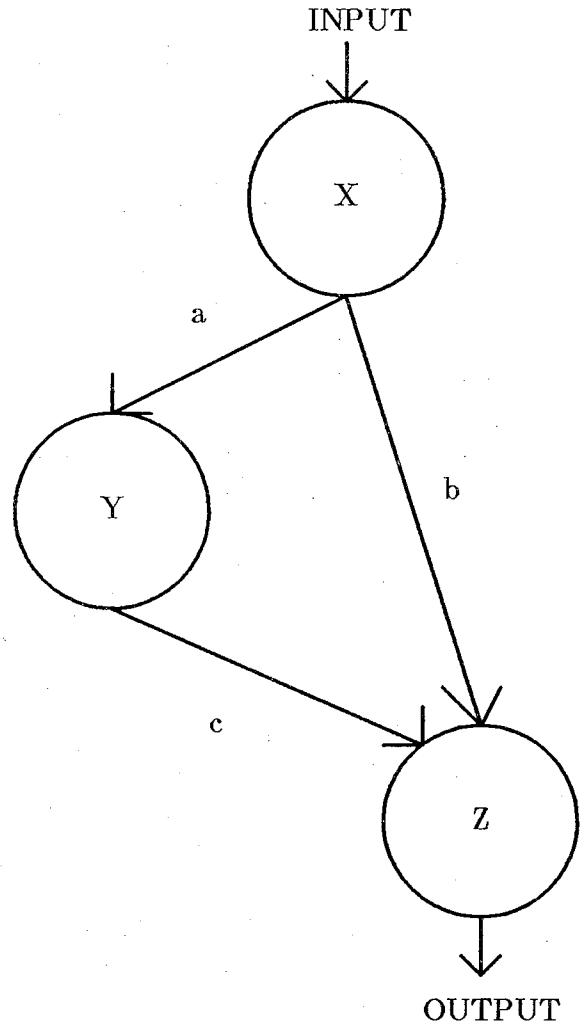


Figure 5.4 Example Data Dependency Graph

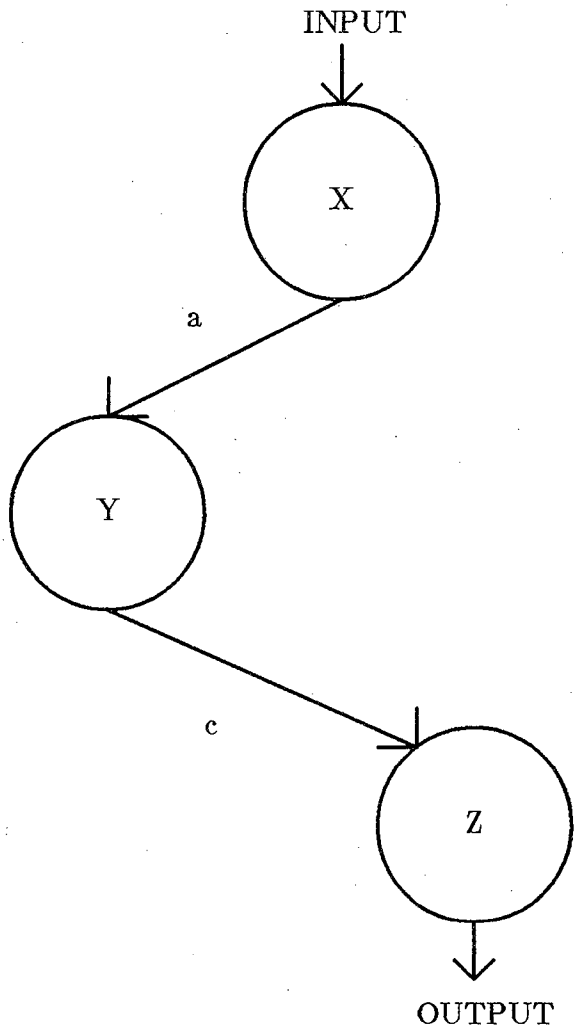


Figure 5.5 Example Reduced Data Dependency Graph

If there exists an edge AZ and a path A B ... Z of length greater than 1, then remove the edge AZ.

In practice, the rule is applied as follows:

For each node in the DDG, build a list of all descendants of the children of that node. Remove the children that are also in the descendant list.

The DDG given for the example task shown in Figure 5.3 contains several redundant arcs. The reduced graph is shown in figure 5.6. The removal of the extra arcs from the "Scene Model" and "Texture Analysis" nodes provide clearer antecedent information and thereby reduces the amount of work that needs to be done by the scheduler.

5.3.3 The DISC System

The expert system that is the major part of DISC makes the scheduling and reconfiguration decisions. There are a number of steps involved in the translation of the RDDG specification of the task to the commands needed by the low-level OS.

At the initialization of the task and at any time during processing that system resources become available, DISC must choose which algorithm in the task should be executed. The decision is based on three factors:

- The amount of processing yet to be done that depends on the execution of a given algorithm. Priority is therefore given to routines that have the largest number of descendents in the RDDG.
- The data that is already contained in a given partition. An attempt is made to reduce the amount of data loading, storage,

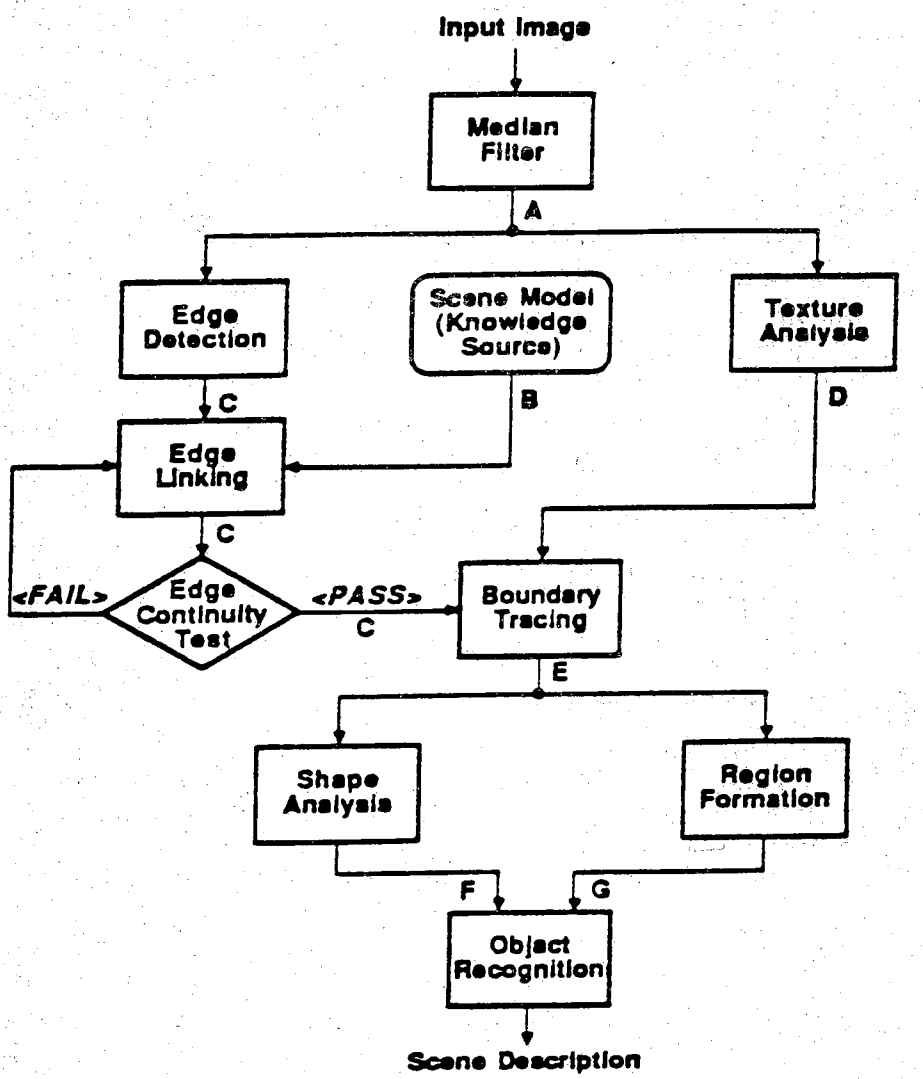


Figure 5.6 Reduced Data Dependency Graph for an Image Understanding Task

and reallocation that is done so that system overhead is kept to a minimum.

- The forced priority of each algorithm. The user (or the IUS) can specify an artificially high or low priority for an algorithm if special processing scenarios are desired.

The number of descendants in the RDDG is an important factor in algorithm selection since it provides an indication of how many other routines depend on the data produced by the candidate algorithm. The ideal indicator would be the amount of later processing, in terms of time and resources, depending on a given algorithm. In practice, this measure is difficult or impossible to calculate. The execution time of an algorithm and its required resources depend on the implementation selected. Since the implementation selection depends on the current machine state, future resource needs and execution times cannot be predicted accurately. Using the number of descendants, however, provides an indication of algorithm importance while adding very little scheduling overhead.

Once an algorithm is chosen to be run, an implementation of that algorithm is selected from the library. One of the major advantages of the DISC system is the ability to choose the algorithm implementation that is most suitable to the present processing scenario and the current state of the parallel processing system. The Algorithm Database (discussed in Section 5.5) contains several different implementations of each algorithm. The distinction might be a different execution mode (SIMD or MIMD), data allocation among the PEs (by row, column, etc.), number of PEs required, etc..

The choice of implementations is based on five factors:

- The mode of the implementation and the mode of the free partition.
- The number of PEs required by the implementation and the number of PEs available in the free partition.

- The relative speedup of the implementation for the given number of PEs.
- The format of the data expected by the implementation and the format of the data in the free partition.
- The data allocation expected by the implementation and the allocation of the data in the free partition.

Also, depending on a number of factors, DISC will either split a free partition into several new partitions, merge several free partitions into a single partition, reconfigure the entire system, or choose the partition that is best suited for the chosen algorithm. These decisions will be discussed in chapter 6.

5.3.4 The Low-Level Operating System Interface

Once an appropriate implementation and system configuration have been chosen, DISC instructs the low-level OS to (1) do any necessary reconfigurations, (2) perform any necessary data loading, reformatting, or reallocating, and (3) begin the execution of the specified implementation. If more system resources are available, DISC repeats the selection process. If no resources are available, DISC monitors the system state until some partition becomes free and then continues scheduling. This process is repeated until the processing required by the task is completed.

5.4 The Low-Level Operating System

The low-level OS has direct control of the parallel processor. It accepts instructions from the IOS and performs the requested system reconfigurations, loads the appropriate implementation code, and performs necessary data I/O. The secondary function of the low-level OS is to monitor the system state and report free partitions to the IOS.

A partition becomes free when an algorithm implementation finishes its execution and releases the PEs used in the partition.

5.5 The Algorithm Database

The algorithm database described here consists of three parts (see Figure 5.7): the Image Understanding System database, the Intelligent Operating System database, and the algorithm library. Note that each of the three components used in executing image understanding tasks uses a part of the algorithm database. It is one of the features of this image processing environment that allows a natural progression from the abstract user's view of the specified task to the specific code and configuration needed to execute a routine required by that task.

5.5.1 Algorithm Library

The algorithms in the database will contain low-, mid-, and high-level image processing routines. The low- and mid-level algorithms are general purpose routines that serve as the basic building blocks for the user's task specification. They are commonly used as preprocessing steps for the high-level routines. Most high-level routines are specific to the given task. An example high-level routine is an expert system designed for object recognition based on region and edge information and a generic scene model.

The algorithm library is the most machine-specific component of the algorithm database. The library consists of different implementations of image processing algorithms. Examples of algorithms to be included in the library are listed in Table 3.3. For each given algorithm, a number of different implementations of that algorithm will be included in the database. As more implementations of algorithms are included in the database, the IOS has the possibility

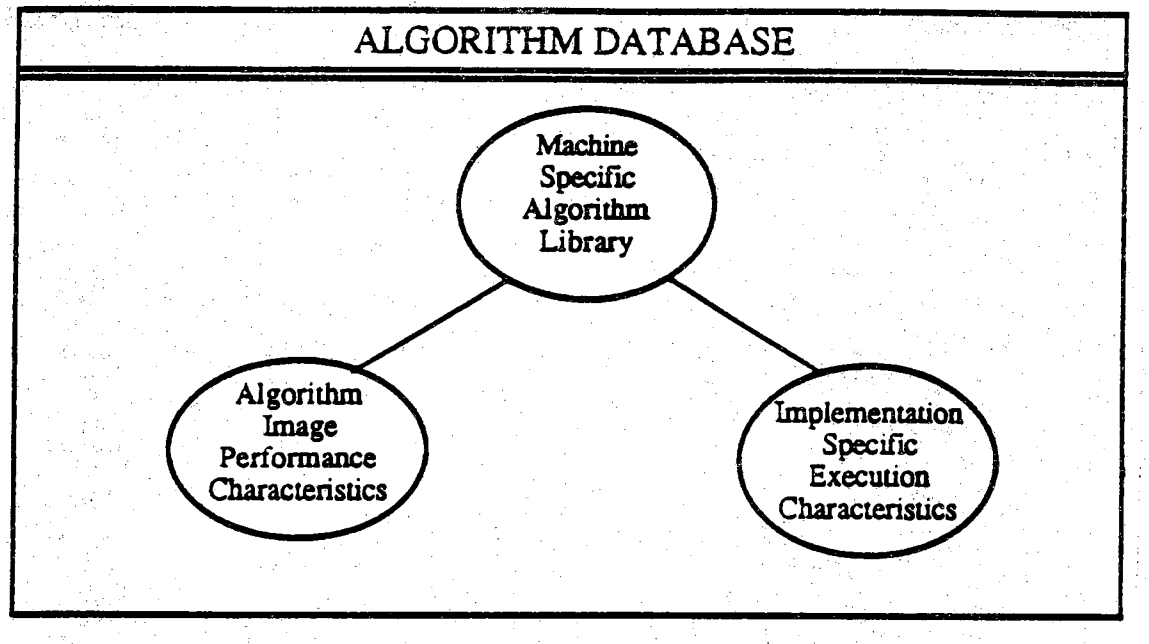


Figure 5.7 Algorithm Database Overview

of constructing more time-efficient schedules since there will be more chances to effectively match processes and partitions. This time savings generally occurs when task migration, data reformatting, and processor reconfiguration are kept to a minimum. Of course, the overall speedup is heavily dependent on routine execution times which are often dependent on the input data.

5.5.2 Image Understanding System Database

The Image Understanding System database contains information on each database algorithm's image analysis performance characteristics. For example, there might be three different edge linking algorithms: one that works well for low noise images, one that works well in noisy images, and one that can deal with badly broken edges. The Image Understanding System uses this data, possibly in conjunction with interaction with the user, in selecting exactly which algorithms will be used to complete a specified task. The Image Understanding System database information, together with the data from the execution characteristics database, also is useful in optimizing the overall task processing time. In some processing scenarios, it may be acceptable to sacrifice resultant analysis performance for a decrease in execution time. For example, an object recognition algorithm may not need a complete set of connected line segments to determine whether or not an object is a building. In this case, a faster but less robust edge linking algorithm may be used. However, it may be important to use the most complete set of edges possible (e.g. recognizing aircraft type from outlines) so that it becomes preferable to use a slower but better edge linker.

Also contained in this database is an algorithm template detailing what the inputs and outputs are for each of the library algorithms. For example, the algorithm *ALG1* might be listed as *ALG1(X,Y,Z)* with X as an **input**, Y as a **modified input**, and Z as an **output**. An **input**

is data that is used by the routine but is not changed in any way. A **modified input** is data that is used by the routine and, upon exit from the routine, has had one or more of its values changed. An **output** is data that is produced by the routine but is not a **modified input.** The distinction between these three types of parameters is important for the scheduling of the routines. A given routine cannot be executed before the data for each of its inputs and modified inputs is available. If the data is produced by another routine (i.e. it is an output of some routine), a time constraint has been placed on the execution order of those two routines. That is, the producer of the data has to finish executing before the consumer can be run.

Figure 5.3 presents a data dependency graph for a "typical" image understanding task. The Image Understanding System might initially represent this task as was shown in example 3.7. The Image Understanding System database contains input and output parameter information for each of these routines. Example 5.1 shows the conceptual form of the template for each routine. The Image Understanding System now has enough information to construct the data dependency graph for the task.

5.5.3 Intelligent Operating System Database

The Intelligent Operating System's function of determining a set of routines to be used to achieve a good execution time for a given task can be compared to the bin packing problem. The scheduling problem here is actually more complicated than "classical" bin packing in that the routines to be scheduled (i.e. the pieces to be packed) are, quite often, of unknown size. Although routines that are not data dependent, such as a 9 by 9 edge mask in SIMD mode, have well defined execution times, data driven routines such as an MIMD edge linking algorithm have execution times that cannot be predicted *a priori.* In order to select an implementation, the IOS will use

Example 5.1 Algorithm Database Template Information

Boundary_trace (INPUT, INPUT, INPUT, OUTPUT)
Edge_continuity (INPUT) : FLOAT
Edge_detect (INPUT, OUTPUT)
Edge_link (INPUT, MODIFIED INPUT)
Median_filter (INPUT, OUTPUT)
Object_recognition (INPUT, INPUT)
Region_Formation (INPUT, INPUT, INPUT, OUTPUT)
Scene_model (INPUT, OUTPUT)
Shape_analysis (INPUT, OUTPUT)
Texture_analysis (INPUT, OUTPUT)

characteristic information stored about each routine.

Each database routine is classified by seven parameters (see Table 5.1). Although it is desirable to have a set of classification parameters that are orthogonal, it is virtually impossible in practice. A more realistic approach is to have the parameters contains enough information for scheduling purposes without an excessive amount of redundancy.

The first parameter is the mode of the algorithm (either SIMD or MIMD). The mode determines whether or not the partition has to be switched from SIMD mode to MIMD mode (or vice versa) for the given routine. Switching the partition mode takes time that cannot be used for processing and therefore should be minimized. This factor may or may not be significant in terms of the task's execution time depending on the amount of system overhead associated with the change. In the PASM system, for example, the system overhead for the switch is

Table 5.1 Database Algorithm Classification Parameters

Algorithm mode (SIMD or MIMD)
Number of processing elements required
Expected execution time
Input data format
Output data format
Input data allocation
Output data allocation

insignificant (only a few instructions) compared to a typical execution time for a routine.

The second parameter is the number of processing elements that are required (or desired) for the routine. In general, the more processing elements used by the routine, the faster the execution time up to some maximum value for that routine. However, processing elements used by one routine are obviously not available for use by another. Some compromise must be made between concurrent execution of routines and execution speed for individual routines. If more processing elements are required than exist in a single idle partition, the system will have to be reconfigured which will increase the overall processing time of the task.

The third parameter is the expected execution time. This parameter is a function of the number of processing elements used, the size of the data set being operated on, the format of the data (binary,

integer, floating point, or symbolic), the mode that the routine is operating in (SIMD or MIMD), and the amount of communication between processing elements. This information is important since it may be more efficient to run a slower routine without reconfiguring the system and/or reallocating the data per processing element than it is to run a faster routine that requires some system reconfiguration. If the execution time of an implementation is unknown, then an internal flag is set in the database and the time is symbolically set to the same constant for all partition sizes. This method of using a constant time value effectively negates any advantage that the implementation would have gained from the relative speedup factor that was discussed in section 5.3.3. DISC understands that this constant indicates unknown time and does not provide an actual execution time estimate.

The fourth parameter is the format of the input data used by the routine and the fifth parameter is the format of the output data produced by the routine. Binary and integer operations are generally faster than floating point operation and therefore should be used whenever possible. Other formats of data include character, double precision, strings of symbolic data, etc.

The sixth and seventh parameters are, respectively, the input and output allocation of data among the processing elements. Typical allocations of data are by rows, columns, and rectangular regions. These two parameters help determine whether or not reconfiguration is necessary. When one routine finishes and the next routine to be executed needs data in the same allocation as that produced by the first routine, it is advantageous not to store the data when the first routine finishes and then reload it when the next routine starts. In this way, overhead from large data transfers can be avoided. The algorithm database also contains specialized routines to perform common reallocation and reformatting functions such as by-row to by-column reallocating and integer to floating point casting. Examples of such routines for a speech analysis task are presented in [Bron82]. These parameters are based on the PEs being processor / memory pairs.

Other configurations, such as PEs connected to memory through an ICN, might not make use of of allocation schemes.

Table 5.2 shows the database information contained for an example implementation of an edge detection algorithm.

These characteristics were chosen since they contain enough information to schedule the algorithms without containing a large amount of redundancy. Adding a parameter to indicate algorithms which spawn new processes might aid in machine partitioning decisions, but extra PEs can be reserved by setting the 'number of PEs' parameter to a large enough value. Other parameters such as algorithm speedup could also have been used by DISC, but adding more parameters will increase the scheduling overhead since each parameter has to be considered when making implementation choices.

5.6 The Knowledge Base

The knowledge base in the environment provides procedural information to the IUS and the IOS. The IUS accesses the knowledge base for task processing schemes. This information could be in the form of the processing needed to complete a given subtask, strategies for algorithm selection, or even expert systems that could be "called" from the IUS.

The knowledge base supplies scheduling strategies to the IOS. In this manner, DISC can be considered to be a part of the knowledge base. Scheduling strategy information might also take the form of a record of how well previous algorithm, implementation, and reconfiguration decisions had performed. This scheme would entail an automatic learning and knowledge acquisition system and is not implemented in the current system. The scheduling strategies used by the IOS are detailed in chapter 6.

Table 5.2 Example *EDGE_DETECT_B* Implementation Data

Parameter	Implementation	
	I1	I2
Mode	SIMD	MIMD
# PEs	$\leq \frac{\#\ \text{pixels}}{64}$	no bound
Time	$\frac{6*\text{image\_size}}{\#\ \text{PEs}} + 4*\#\ \text{PEs}$	$\frac{4*\text{image\_size}}{\#\ \text{PEs}} + \frac{\text{image\_size}}{64}$
Input format	1 byte/pixel	1 byte/pixel
Output format	edge list	edge list
Input allocation	regions	regions
Output allocation	regions	regions
Parameter	Implementation	
	I3	I4
Mode	SIMD	MIMD
# PEs	$\leq \frac{\#\ \text{pixels}}{64}$	no bound
Time	$\frac{6*\text{image\_size}}{\#\ \text{PEs}}$	$\frac{4*\text{image\_size}}{\#\ \text{PEs}} + \frac{\text{image\_size}}{64}$
Input format	1 byte/pixel	1 byte/pixel
Output format	binary image	binary image
Input allocation	regions	regions
Output allocation	regions	regions

5.7 Task Execution Sequence

To tie all the environment components together and summarize the processing done during the execution of a task, the steps involved running a user's task on the parallel processing system will be recapped.

When an image understanding task is run on the parallel processing system, the algorithm database is used in the process of expressing the task in a form usable by the low-level operating system. The first step is for the IUS to translate the user's task specification into a set of algorithms that need to be run to complete the task. This set will be generated in the form of a data dependency graph. This step makes use of the Image Understanding System database when selecting which algorithms will be used. Speed/performance tradeoffs can be made to tailor the algorithm selection to the current application.

The IOS then reduces the DDG by removing redundant data arcs from the graph. Once the graph reduction is complete, one of the implementations of each algorithm must be chosen by using the stored database information about each routine and by taking into account the current operating environment. Exactly which implementation of an algorithm the IOS choses is based on trying to minimize the overall execution time of the specified task. The data contained in the Intelligent Operating System database (i.e. algorithm execution characteristics) allows the system to fit the best implementation of the specified algorithm into the overall processing scenario. Factors to be taken into account in the routine selection include the current partitioning of the parallel processor, the routine executed prior to the current one, the routines that will be executed after the current one, reconfiguration overhead, and task migration overhead.

From the reduced data dependency graph, the scheduler determines a partitioning of the parallel processor and which routine will be run in each partition. Assuming that all the algorithms for a given task cannot be run concurrently on the parallel processor, this scheduling process is dynamic. As processes finish executing and

intermediate information becomes available, the scheduler must determine the next configuration and set of routines to be run.

The system configuration and choice of routines to be run are passed to the low-level operating system. The parallel processor is then repartitioned (if necessary) and the code for each routine is loaded from the algorithm library and placed in the appropriate partition. This process is repeated until all processing required by the task is completed.

CHAPTER 6
DISC DESIGN CONSIDERATIONS

This chapter presents a detailed analysis of the implementation of the DISC system. The first two sections discuss the externals of the system: the implementation language and the code layout. The next section discusses the heuristics used in the DISC system. The last section presents details of the code used to implement those heuristics. Appendix C provides a discussion of the formats and contents of the various code sections used in DISC.

6.1 DISC Implementation Language

The DISC system is implemented as a rule-based expert system. There were several considerations that influenced that decision. This section discusses both the choice to use a rule-based system and the chosen implementation language.

6.1.1 Language Considerations

One of the major reasons for choosing an expert system approach is the lack of any standard algorithms for scheduling jobs with unknown execution characteristics. This situation leaves the possibility of three strategies. The first possibility is to develop a deterministic

algorithm that can deal with unknown execution times. This strategy is probably infeasible since algorithms with unknown execution characteristics are, by their very nature, nondeterministic and since the reconfiguration possibilities grow exponentially with machine size (this subject will be discussed later). The second possibility is to use the available scheduling techniques (such as simulated annealing) as much as possible and deal with the unknown cases separately. There are two problems with this scheme. The first problem is that the nondeterministic algorithms still have to be dealt with even if a good schedule for the deterministic ones can be obtained. The second drawback is that there would very likely be a large amount of wasted system overhead due to scheduling deterministic sections of the task that, due to branching in the dependency graph, are either not executed or are executed with a different system state than is initially planned for. The last possibility is to use a heuristic-based system to apply intelligence to the scheduling problem.

An expert system (*ES*) is, by definition, the natural choice for implementing a heuristic-based system. Since the inference engine is built in to the expert system shell, it has the advantage of being unaffected by the task size. The task looks the same to the ES whether it contains 1 algorithm or 100 since the ES shell takes care of applying each heuristic to every algorithm.

At this point, it will be helpful to clarify the distinction between *expert system*, *expert system shell*, and *inference engine*. The expert system is comprised of the rules that implement heuristics. The expert system shell has three functions. It acts as a user interface, interprets and applies the rules in the ES, and manages the facts used by the ES. The application of the rules to the facts contained in the fact base is accomplished by the inference engine. Conceptually, it choses a rule whose conditions for firing have been met by the fact base, fires the rule, and makes any necessary changes to the fact base. This process is continued until no more rules can be fired.

As mentioned above, the possible machine states for a given reconfigurable architecture grow exponentially with the number of PEs and with the number of partitions. For example, consider a reconfigurable parallel processor with 32 micro controllers (such as PASM). The parallel processor is therefore capable of having any number of partitions from 1 to 32. For a given number of partitions, the machine is capable of being in a variety of states since configurations are not isomorphic. That is, the machine configuration

((0 16 8 24 4 20 12 28 2 18 10 26 6 22 14 30)
(1 17 9 25 5 21 13 29) (3 19 11 27 7 23 15 31))

is distinct from the configuration

((0 16 8 24 4 20 12 28) (2 18 10 26 6 22 14 30)
(1 17 9 25 5 21 13 29 3 19 11 27 7 23 15 31))

even though both have three partitions with equivalent sizes. (The MCs contained in each partition for this example conform to a hypercube assignment.) The non-equivalence of the configurations is due to differences in partition data contents and algorithm execution status. If no data was loaded and no algorithms were executing, the two partitions would be functionally equivalent. Appendix A discusses this problem in depth. To make this idea more concrete, the PASM machine can have 100 distinct machine states with 7 partitions and 55308 distinct machine states with 21 partitions. Since overhead time has to be kept to a minimum, the scheduler cannot consider every possible case. An expert system, however, can apply heuristics to quickly find an acceptable but possibly sub-optimal solution.

An ES is ideal for system prototyping. The heuristics are easy to modify since they are bundled into separate rules. Adding new heuristics or modifying existing ones is therefore relatively straightforward.

One disadvantage of using expert systems is their interpretive nature. The ES shell must process rules as they are fired. This procedure is slower than having the rules execute directly on the underlying hardware. Some ES shells have the capability to compile the rules into an executable form, and this ability is a factor in choosing a suitable language.

6.1.2 The CLIPS Expert System Shell

The language that is used for the implementation of DISC is the CLIPS (C Language Integrated Production System) expert system shell developed at NASA [Culb88, Giar88]. CLIPS has several features that influenced its selection.

The ES code can be directly linked with the operating system through built-in system access functions. User written C code can be compiled into the system so that computationally intensive functions can be processed in an efficient manner. C code has the added benefit of being able to do low-level system access. These C routines can then be accessed in the same manner as built-in functions.

Expert system code that has been written for the CLIPS shell can be translated into C code through a mechanism inherent to CLIPS. This C code can then be compiled to execute without the intervening shell interpreter. This process can lead to a substantial reduction in the amount of system overhead used by the ES.

As a minor consideration, the code for the CLIPS shell (itself written in C) is available so that detailed profiling information can be obtained for ES runs. Also, the shell contains various status and debugging commands. Both of these features have aided in the prototyping of the DISC system.

6.2 DISC Code Layout

The code for DISC is split up into several sections roughly corresponding to functional divisions. Table 6.1 lists these segments and their major purposes. The DISC implementation also contains several C routines and external data files. Throughout the code, every attempt was made to minimize hardware dependencies. The prototype parallel processing system is PASM and the code had to be tailored for that architecture. However, information specific to PASM such as the number of PEs and the possible interconnection network configurations are read at initialization time from parameter files whenever possible.

Table 6.1 DISC Code Sections

Segment	Purpose
system initialization	load and process the DDG
database load	load and process database information
main part 1	choose an algorithm and machine configuration
main part 2	choose an implementation of the algorithm
main part 3	start the implementation execution
main part 4	process finished implementations
system cleanup	prepare for the next task

The system initialization code prepares the DISC system for the execution of the first task. Several steps are involved in the startup.

First, the fact base is initialized with the default processor configuration (all PEs in a single partition). The data dependency graph is then read and processed (through external C functions) and the resulting reduced graph is read into the fact base. Next, a list of which algorithms are immediately executable is built. Finally, precedence information in the form of algorithm descendant lists is extracted from the RDDG.

Once initialization is complete, the database information for any algorithms not previously loaded is read in. Parameters for the task are then loaded from an external file and the symbolic equations for the execution times of the algorithms are evaluated and stored in numeric form. Once these steps are completed, the scheduling can begin.

The main scheduling loop is divided into four sections. The first section is in charge of devising new machine configurations and choosing which algorithm will be run next and in which partition. Possible repartitioning is decided based on the number of free partitions in the machine at the current time (there must be at least one to be at this point in the processing) and the number of algorithms that could possibly be run at the current time (again, there must be at least one). The algorithm is selected based on three factors:

A) The number of descendants of the candidate algorithm. The number of descendants an algorithm has indicates the amount of future processing depending on it. As descendants increase, the priority of an algorithm is increased.

B) Partition data contents. If some free partition contains the data needed by the candidate algorithm, it is more likely that that algorithm will be chosen.

C) The user specified priority value of the algorithm. This factor is actually the overriding condition if some algorithm has a priority higher than all others.

Once the algorithm has been selected, a partition is chosen based on using the largest partition available and taking into account data matching between the partition and the algorithm.

The second step in the main loop chooses a database implementation of the chosen algorithm. The choice is based on five factors:

A) The mode of the chosen partition versus the mode of the candidate implementation. The value of the match is weighted by the amount of system overhead involved in a mode switch.

B) The number of PEs required for the candidate implementation compared with the number of PEs in the chosen partition. No weight is used here, but an implementation can be rejected entirely if the partition is not of an appropriate size.

C) The execution time for the candidate implementation given the number of PEs in the partition. This value is weighted by the speedup over the slowest possible case (using the minimum number of PEs). As was previously mentioned, implementations with unknown execution times are unaffected by this measure since the weight is 1 and DISC treats all candidates as though they had the same execution time.

D) The format of the data in the partition compared to the format of the data required by the candidate implementation. The comparison is weighted with the amount of time it would take to convert the data to the proper form.

E) The allocation of the data among the PEs in the partition compared to the allocation of the data required by the candidate implementation. The comparison is weighted with the amount of time it would take to reallocate the data.

The third DISC loop step informs the low-level operating system to start the execution of the chosen implementation. It instructs the

low-level OS to do any data loading, reformatting, or reallocating necessary and then to begin the execution of the implementation.

The last DISC loop step waits until some partition becomes free and then processes the information from the finished implementation. It updates the list of algorithms that can currently be executed by adding children of the finished algorithm that have all input data available. If the finished algorithm is a conditional (a branching node in the DDG), the results of the condition are used to determine what processing can be done next. That is, the appropriate branch of the DDG is activated.

The system cleanup code executes when the entire task has been completed. It removes unnecessary facts from the fact base so that subsequent tasks to be processed are not affected by previous tasks. It also checks to see if a new task is available and, if so, begins the processing of that task. The system then goes back to the 'database load' portion of the code and repeats the processing steps.

DISC uses external C functions for three basic purposes. First, the user's task in DDG form is read in, reduced to RDDG form, and asserted into the fact base. The second function is to evaluate the execution time equations from the algorithm database. The third purpose is to compute new machine configurations. Since expert systems are not well suited to these kinds of processing (graph traversal, numeric processing, and graph manipulation), the C code simplifies and speeds up the DISC system.

6.3 DISC Heuristics

DISC employs two different types of heuristics: those used for scheduling purposes and those used for machine repartitioning. Although they are not totally separate in the code, they are functionally distinct and can be discussed separately. This section discusses the heuristics used, Section 6.4 details their implementation.

6.3.1 Scheduling Heuristics

The overall purpose of the scheduling heuristics is to see that the user's task is completed in an efficient manner. Due to the way DISC processing is done, the scheduling entails choosing an algorithm to run, choosing the appropriate free partition, and then choosing one of the implementations of that algorithm. These three steps are separate in the code and are done sequentially.

The underlying philosophy of choosing an algorithm is to consider all algorithms that are capable of being executed at that time and picking the one that is most appropriate for the overall processing scenario and the current state of the parallel processor. The first step is to consider only those currently executable algorithms which have the highest user specified priority. A voting scheme is set up in which each heuristic can express its approval (or disapproval) of each candidate. The algorithm with the highest score is then chosen.

The first heuristic considers the number of descendants an algorithm has in the RDDG. The vote for the algorithm is its number of descendants. Loops in the RDDG are handled by multiplying the number of descendants by a loop weighting factor. In this manner, the fact that a loop can execute more than once is acknowledged. The optimal method would be to apply probabilistic methods to the loop to determine an expected number of passes through the loop. However, there are several factors involved and this method could significantly add to the system overhead time. The loop weight factor is therefore a compromise measure.

The next heuristic votes based on matching partition data contents and algorithm data requirements. The vote is incremented by 1 if some candidate's inputs match some free partition's data. This number is the smallest possible increment because it is not guaranteed that the algorithm will be executed in the matching partition. The vote is used as a tie-breaking measure.

The partition to be used is chosen on two factors: size and data. This heuristic will choose the free partition with the largest number of PEs and break ties by also trying to match data contents to algorithm input data. At this point, an implementation of the selected algorithm can be chosen.

The implementation is chosen based on attempting to complete the processing required for the selected algorithm in the most efficient manner. As with algorithm selection, a voting scheme is initiated.

The first heuristic compares the partition mode and the implementation mode. The weight for this vote is the amount of system overhead time used by the mode switch operation.

The second heuristic checks if the partition has the proper number of PEs for the implementation. Too few (or too many) PEs can cause the immediate rejection of the implementation if such a restriction occurs in the database information for the implementation.

The next heuristic considers the relative speedup given the number of PEs in the partition. That is, the vote is the ratio of the execution time for the selected partition size to the execution time for the minimum partition size. In this manner, precedence is given to the implementation which will benefit most from the given number of PEs.

The last heuristic compares the data format and allocation in the selected partition to the data format and allocation required by the candidate implementation. The amount of time required to reformat and/or reallocate the data is considered as a detraction from the candidate implementation.

The votes are then compared and the highest scoring implementation is chosen for execution.

6.3.2 Repartitioning Heuristics

The decision of when and how to repartition the parallel processor presents a difficult problem. As stated earlier, the number of possibilities increases exponentially with machine size and number of partitions. For example, consider a ring-connected machine with 4 PEs (0, 1, 2, and 3) in which any adjacent PEs can form a partition. Assume partitions 0 and 2 are actively running algorithms and partitions 1 and 3 are idle. There are then five unique ways to partition the machine. To maintain four separate partitions, the machine is left in the same state: ((0) (1) (2) (3)). Also, there are four different ways to merge two idle partitions and leave two single partitions for the active algorithms: ((0 1) (2) (3)), ((0) (1 2) (3)), ((0) (1) (2 3)), and ((0 3) (1) (2)). The best choice would be a function of the repartitioning time, the amount of time it takes to move the active jobs to new partitions, and the current processing needs.

Because of the intractable nature of the full repartitioning problem, a totally different approach is employed. The method used is a variation of the operating system concept of *compaction* [Aho86]. When the machine partitions become *fragmented* (that is, there are unmergeable free partitions), the system can be compacted to maximize free partition size. The details of this process are explained in Section 6.4.

Table 6.2 outlines the basic repartitioning strategy. Some of the possibilities are intuitively obvious. For example, if there are no partitions free or there are no algorithms that can be executed, then there is nothing to do but put the scheduler into a wait loop. If there is only one algorithm that can be executed and there is only one free partition, then the heuristic is to run that algorithm in the given partition. The rest of the heuristics are driven by the following observation: it is almost always more efficient to run multiple algorithms simultaneously on smaller partitions than to run them sequentially on a single, larger partition. This phenomenon is caused

Table 6.2 Repartitioning Strategy

		Number of Executable Algorithms		
		0	1	N (N > 1)
Number of Free Partitions	0	wait	wait	wait
	1	wait	run it in the partition	split into multiple partitions
	P (P > 1)	wait	merge if the partitions are adjacent	compact the system if P != N

by the increased overhead of inter-PE communication costs. That is, there is a nonlinear speedup as the number of PEs increase.

There is a need to balance never having an idle partition with the cost of repartitioning the system. The following heuristics are used to implement a simple yet effective strategy.

The first heuristic handles the case of one free partition and multiple algorithms that can be run. It states that if the partition is already as small as possible, then use that partition and continue scheduling. If the partition can be split into multiple partitions, then split it into the appropriate number of new partitions and continue scheduling. The appropriate number of new partitions, P′, is

$$P' = \min(N, S)$$

where N is the number of algorithms that can be currently executed and S is the number of micro controllers in the partition. In other words, S is the maximum number of new partitions that can be created from the free partition.

The next heuristic deals with the case of multiple free partitions and only one algorithm that can be run. If some of the partitions are mergeable, then the largest possible partition is made. If none of them are adjacent, then scheduling continues. Algorithms are not always given the maximum number of PEs by this scheme, but active partitions are not disturbed. In this manner, the only overhead expended is in the actual partition merge operation.

The last heuristic handles the situation of multiple free partitions and multiple algorithms that can be executed. In the case that the number of free partitions equals the number of executable algorithms, the system is left in the present state. In the other case, the system is compacted and P′ partitions are created where P′ is as previously defined. Briefly, compaction involves restructuring the machine configuration so that the largest possible free partitions are created. The split, merge, and compaction operations are discussed in detail in the next section.

6.4 DISC Heuristic Implementation Details

This section will outline the operation of DISC from a rule level instead of the heuristic level of the previous section. The specifics of the reconfiguration routines will also be detailed.

6.4.1 Rule Execution Ordering

Expert systems are inherently non-sequential since the inference engine takes no account of rule ordering. Given that two rules can both be fired, the rule that is chosen might be based on how much time has passed since the rule last fired, which rule has been ready to fire for the longest time, how many facts were involved in the rule activation, or many other schemes. There are two recourses when sequential action is desired. The first possibility is to make the rules dependent on some specific fact and to add and remove that fact at the appropriate times. This method increases the complexity of the fact base and makes extra work for the inference engine, thereby slowing down the system. The second method is to add priority information to the rules. In this way, two activated rules with the same priority will execute in an arbitrary order but the rule with the higher priority will be chosen if the priorities differ. This method places no added burden on the inference engine.

DISC does some processing that is inherently sequential and some in which processing order is inconsequential. The scheduling steps must be done in the order already discussed. However, the order in which algorithms, implementations, and partitions are voted on is not important. The problem of sequential processing in DISC was solved by giving each rule a priority value. When firing order is unimportant, rules have identical priority values.

6.4.2 Processing Lists

DISC maintains several lists during processing. The first list is *DB_LOADED* which keeps track of which algorithm databases have already been loaded. This list prevents reloading information the system already contains when subsequent tasks are processed.

The *FEX* list (*F*inished *EX*ecuting) keeps track of which algorithms have already been run. This information is important since algorithm precedence is based on data generation and usage.

The *ACE* list (*A*lgorithm *C*urrently *E*xecuting) keeps track of which algorithms are currently being processed on the parallel processor. This list is mostly for bookkeeping purposes.

The *PEN* list (*P*otentially *E*xecutable *N*ow) tabulates which algorithms are candidates for immediate execution on the parallel processor. Subordinate to this list is the *PENMP* list (*PEN M*aximum *P*riority) which keeps track of those PEN algorithms that have the highest user specified priority value. It is from the PENMP list that algorithms are chosen for execution.

There are also several smaller lists maintained. These lists include the parents of each algorithm, the descendants of each algorithm, and the size, mode, and contents of each machine partition.

6.4.3 Configurations, Splitting, and Merging

DISC maintains information in its fact base about possible machine configurations. A complete tabulation of every possible configuration would be prohibitively large, so information on how reconfigurations can be done is listed instead. For example, instead of listing that the following partitions are possible:

(0), (1), (2), (3),

(0 1), (2 3), (0 1 2 3)

the information on mergeability is indicated. That is, it indicates that partitions (0) and (1) can be merged, (2) and (3) can be merged, and (0 1) can be merged with (2 3):

$$(0)\ (1) \rightarrow (0\ 1)$$
$$(2)\ (3) \rightarrow (2\ 3)$$
$$(0\ 1)\ (2\ 3) \rightarrow (0\ 1\ 2\ 3)$$

This information can be easily modified for different underlying architectures.

When DISC wants to form a larger partition, it can repeatedly apply the merge rules until a partition of the desired size is obtained. By reverse application of this process, the splitting of partitions can be accomplished. These procedures cause very little system overhead since the strategies are coded into rules. That is, if no partitions can be split or merged, no overhead is spent looking for possibilities since no facts will activate these rules.

When a partition is split into smaller partitions, the following algorithm is used. The strategy is based on reconfiguration rules such as are used in the PASM system. The restrictions are that partitions must contain a power-of-2 number of PEs and that all MCs grouped into a partition of size 2^p must have identical low-order 10—p bits in their physical addresses. Let N be the number of desired partitions. Let S be the largest number of new partitions into which the partition can be split. Let T be the largest power of 2 less than or equal to N. Let P be the number of PEs in the partition.

One of two conditions must be true: either $N \geqq S$ or $N < S$. When $N \geqq S$, more partitions are desired than can be generated so the partition is simply split into as many new partitions as possible. When $N < S$, it is desirable to split the number of available PEs as evenly as possible among the N new partitions (based on the previously

stated heuristic that it is more efficient to run as many algorithms simultaneously as possible). For this case, DISC will make (2T−N) partitions of size $\frac{P}{T}$ and 2(N−T) partitions of size $\frac{P}{2T}$. Note that the total number of new partitions is then

$$(2T-N) + 2(N-T) = N$$

and that the total number of PEs in the new partitions is

$$(2T-N)\left(\frac{P}{T}\right) + 2(N-T)\left(\frac{P}{2T}\right) = P$$

as are expected.

6.4.4 System Compaction

System compaction is a relatively expensive operation in terms of overhead [Schw88] and so is performed only when simple partition splitting or merging is not sufficient for the current processing needs. In system compaction, the active partitions are packed together so that idle partition sizes are maximized. This scheme differs from the memory compaction used in operating systems in that partitions can be reordered instead of linearly moving them together. The reordering becomes necessary because of restrictions placed on partition formation by the interconnection network.

In order to achieve the optimal packing (no free partitions surrounded by active ones) in the minimum overhead time, the jobs in partitions are only moved if it is absolutely necessary. Assuming a control hierarchy such as is employed in PASM, the following algorithm will achieve the desired packing.

The active partitions are processed from largest to smallest. This scheme achieves optimal packing since the partitions will then always

fall on the proper ICN boundaries. The active partitions are packed into the leftmost part of the machine (assuming a planar tree structure). The same algorithm will work equally as well moving active partitions to the rightmost part of the machine, though. The choice of leftmost is arbitrary.

The compaction algorithm is as follows:

1) Sort the active partitions by size, largest first.
2) Mark the active partitions that are already in place. The final location of any partition can be calculated since the size of every partition is known.
3) Going from largest to smallest, move each active partition to the leftmost location into which it will fit.

After the compaction, all free partitions are grouped together at the rightmost part of the machine. The partitions can then be segmented into the desired number of new partitions. The algorithm for configuring the free partitions is as follows. Let N be the desired number of partitions and let S be the maximum number of partitions obtainable.

1) If $N \geqq S$, then make S partitions and exit.
2) Make the free partitions as large as possible by doing all possible merges.
3) If $N \leqq S$, then exit.
4) If every free partition is as small as possible, then exit, Otherwise, split the largest free partition in half.
5) If $N = S$, then exit.
6) Go back to step 4.

Steps 2 and 4 of the preceding algorithm are conceptual only. The

actual machine does not have to be merged and repeatedly split since only the final state is actually used.

Figures 6.1 and 6.2 illustrate the results of a compaction operation. The example machine has 16 PEs numbered from 0 to 15 (shown in []'s under the figures). The partitions are numbered in the boxes and those partitions that are executing algorithms have an "A" under the partition number. In the uncompacted state, there are six active partitions: one of size 4, two of size 2, and three of size 1. There are 4 idle partitions: one of size 2 and three of size 1. After the compaction operation, all five idle PEs are adjacent in the machine. Depending on the number of new partitions desired, these idle PEs can be grouped as needed. All active partitions have been moved to their new locations. Note that partition 3 did not have to be moved since its original location already conformed to the new packing.

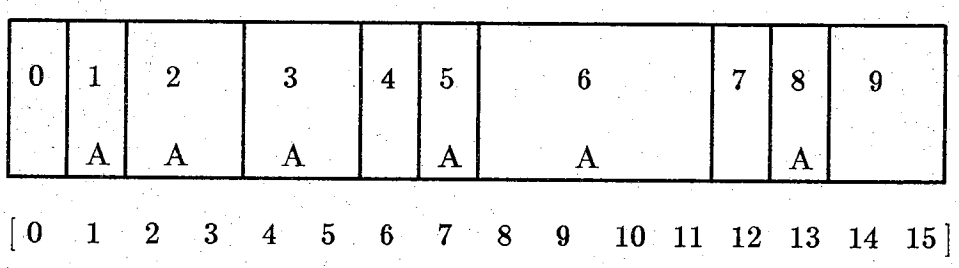


Figure 6.1 Machine in Uncompacted State

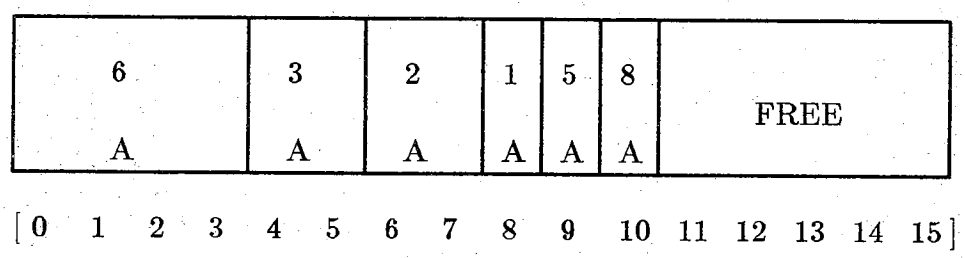


Figure 6.2 Machine in Compacted State

CHAPTER 7
DISC PERFORMANCE MEASURES

This chapter will present a discussion of performance measures for parallel processing systems. Criteria for evaluating algorithm performance are presented in the first section. In the second section, criteria for evaluating task performance are discussed. The third section presents representative results from tasks processed by the DISC system.

7.1 Algorithm Performance Criteria

A large number of criteria can be used when measuring the performance of individual algorithms on a parallel processing system [Lee80, Sieg82]. There is no one "best" method of determining how well an algorithm is executing on the parallel processor. In some cases, the total execution time may be of critical importance while in other instances the processor utilization may be more important. In many cases, it may be most appropriate to measure the performance by all possible standards to get an overall view.

Several common algorithm performance criteria are listed in Table 7.1. This list is by no means complete, but it does present a diverse overview of the types of performance criteria that can be used.

The *Compression* [Lee80] is the ratio of the number of operations executed in an equivalent serial algorithm to the number of operations

Table 7.1 Algorithm Performance Criteria

Compression
Efficiency
Execution Time
Overhead Ratio
Parallel Index
Redundancy
Speed
Speedup
Utilization

executed in the parallel algorithm. A high compression measure means that a small amount of extra computations have to be done to gain the extra speed. This measure is the inverse of *Redundancy.*

The *Efficiency* [Lee80, Sieg82] is defined as the *Speedup* of the algorithm divided by the number of PEs used. It gives an indication of how effectively the PEs are being used to achieve the increased speed. The *Efficiency* follows the intuitive idea that it is best to gain speed with as few processors as possible.

The *Execution Time* [Sieg82] is simply the amount of time spent processing the algorithm. This parameter gives an absolute measure of how well the algorithm is performing, but does not give any indication of how well it could potentially be performing.

Overhead Ratio [Sieg82] is the ratio of the amount of overhead time to the total processing time. It gives an indication of the relative amount of time spent actually performing the algorithm computations.

The *Parallel Index* [Lee80] is a measure of the average speed of the algorithm. It is defined as the ratio of the total number of operations executed (counting operations done in parallel as separate operations) to the total execution time in steps. It provides the average number of operations that are done in parallel.

The *Redundancy* [Lee80, Sieg82] of the algorithm is defined as the inverse of *Compression.* A high redundancy means that the speed increase is being gained at a high price in terms of the amount of computations that are done.

The *Speed* [Sieg82] of an algorithm is a raw measure of the amount of data that is being processed in one unit of time. It can be defined as the total amount of data processed divided by the total *Execution Time.*

The *Speedup* [Lee80, Sieg82] measures how much faster the algorithm will execute on the parallel machine than on a serial machine. That is, it is the execution time for 1 processor divided by the time for an N PE parallel machine. The closer this number is to N, the closer the algorithm is to being ideal.

Finally, the *Utilization* [Lee80, Sieg82] is a measure of the amount of time the PEs are actually involved in the computations of the algorithm. It is defined as the average amount of time each PE executes algorithm operations as opposed to being idle due to disuse or system overhead.

Although these criteria can effectively characterize the performance of individual algorithms, they are of limited use in the characterization of entire tasks on reconfigurable machines. The problem stems from the fact that it is generally difficult or impossible to determine the optimal schedule for comparison. If an optimal schedule were available, there would be little point in testing other

strategies.

7.2 Task Performance Criteria

Since the DISC system deals with entire tasks, criteria were devised to give an indication of overall system performance. Some of the performance measures for tasks are analogous to those for algorithms. For example, the overhead involved in scheduling and reconfigurations can be used in the manner that communication overhead is used for algorithms. However, the general lack of a baseline schedule makes relative performance measures more difficult to define. This section will discuss some of the possibilities for task performance measures.

Table 7.2 lists some of the performance criteria that can be used for tasks. Several similarities exist between these measures and those for algorithm performance.

The *Average Time Per Algorithm* for a task τ (A_τ) measures the average execution time for each algorithm in the task. It is defined as

$$A_\tau = \frac{T_\tau}{\alpha_\tau}$$

where T_τ is the total execution time for task τ and α_τ is the number of algorithms executed in the task. This criterion is useful for comparing tasks that have the same input data and use the same algorithms in a different order. In general, a smaller A_τ indicates better performance.

The *Tiling Percentage* of a task (P_τ) is defined as 1 minus the ratio of partition idle time to task execution time:

$$P_\tau = 1 - \frac{I_\tau}{T_\tau}$$

where I_τ is the total partition idle time for the task. Since the number

Table 7.2 Task Performance Criteria

Average Time Per Algorithm
Tiling Percentage
Task Execution Time
Scheduling Optimality
Scheduling Improvement

of partitions can change during the processing of a task, P_τ is defined in terms of areas. Consider a rectangle with one dimension being the total number of PEs in the system and the other dimension being execution time. The complete rectangle for a task then has an area of $N_M\ T_\tau$ where N_M is the total number of PEs in machine M. The idle time of a partition in terms of areas is $N_\rho\ T_\rho$ where N_ρ is the number of PEs in the idle partition ρ and T_ρ is the amount of time that partition ρ is idle. P_τ can then be defined as

$$P_\tau = 1 - \frac{\sum N_\rho\ T_\rho}{N_M\ T_\tau}$$

where the summation is over all occasions when some partition is idle. *Tiling Percentage* can be a misleading term since smaller idle time does not necessarily mean smaller task execution time. There may be times when expending some system overhead will result in a reduced task time. In this case, P_τ will be larger than for the slower task. However, it is an indication that potential inefficiencies exist. Values for P_τ can

range from 0 to 1 with 1 being the best.

The *Task Execution Time* is, in general, the factor that the system is trying to minimize. It is simply the total amount of time needed to complete the entire task. The value can be given either in real time units or in instructions processed. The main problem in using this criterion is that there are no metrics to rate the merit of the numbers obtained. This parameter can be used, however, to rate the relative speed of the same task processed in different ways.

The *Scheduling Optimality* is defined as the task execution time for the best possible schedule divided by the actual task execution time:

$$O_\tau = \frac{T_{\tau,\text{best}}}{T_\tau}$$

In general, the best possible case cannot be determined. This parameter is useful in those situations where the optimal schedule is known beforehand and new scheduling strategies are being compared. Values for O_τ range from 0 to 1 with 1 indicating the optimal schedule.

The *Scheduling Improvement* is a comparison of the worst possible task execution time and the actual execution time. It is defined as

$$S_\tau = \frac{T_{\tau,\text{worst}}}{T_\tau}$$

The *Scheduling Improvement* is a measure of how much better the task is being processed than the worst possible case using the same algorithms. The worst possible time is defined as worst time in parallel and not the serial time. One way to get a number for the worst time is to assume a naive scheduling strategy in which all algorithms are executed sequentially using all resources for each algorithm. Although this idea makes intuitive sense, it will not be possible to get a number for the worst case for many tasks since at least one algorithm in the task will probably have a nondeterministic execution time.

By using the performance criteria that can be applied to a given task, a qualitative indication of overall scheduling performance can be obtained.

7.3 DISC Results

This section presents a performance analysis of the DISC system. Example results from simulation runs are shown and timing information is given. A discussion of the relation of the simulation runs to actual executions is also presented.

7.3.1 Simulations

For the simulation runs as well as for the use of DISC on an actual parallel processor, the quality of the scheduling partly depends on the richness of the algorithm implementation database. Increasing the number of algorithm implementations allows DISC more opportunities to find good matches between processing needs and available implementations. For the simulation runs used in testing DISC, 2 to 3 implementations of each algorithm were in the database. Each implementation had code for all of the 6 legal partition sizes on PASM (32, 64, 128, 256, 512, and 1024 PEs). Although the codings for different partition sizes affectively give 12 to 18 implementations for each algorithm, the size of the candidate partition dictates which implementations are capable of being used. Once a partition size is determined, the implementation choice is restricted to the 2 or 3 different implementations for the given size. Appendix C contains a listing of an example algorithm database file (listed as db.region_formation).

For a given partition size, the algorithm implementations differed by one or more of the classification parameters. For example, the two

implementations of the edge continuity algorithm differ in mode (on is SIMD, one is MIMD), number of PEs required (the SIMD version restricts the number of PEs in the partition to be less than or equal to the number of pixels per row in the image, the MIMD version does not place a restriction on the number of PEs), the execution time for a given partition size, and the input allocation of the data (by row for the SIMD version, by column for the MIMD version). In this manner, DISC has to choose between implementations that are the same in some respects and different in others.

It was decided that simulations would be used to test the performance of the DISC system. Actual execution runs were deemed infeasible for a number of reasons. First, the target parallel processor, PASM, is currently still in the development stage. It is presently configured with 4 MCs, each with 4 PEs. Having a maximum of 4 partitions severely limits the variability of the system and does not permit extensive testing. Also, the low-level operating system does not yet have sufficient power and interface capabilities to be useful to DISC. Second, actually running hundreds of tasks could take a prohibitively long amount of time even on a parallel processing system. Third, detailed low-level timing information is needed for the analysis and this information is either unavailable on the parallel processor or would artificially slow the execution. Finally, parallel implementations of all the library algorithms have not yet been entered into the system.

The results from simulations are independent of the power of the underlying processor that executes the DISC code and can therefore be extended to different machines. The drawbacks of simulations are that it can be difficult to relate simulation time to real time and that the processing steps are slightly different to account for the fact that the simulator has to keep track of the machine state instead of simply querying the low-level operating system.

Testing of the DISC system was performed in two phases. The purpose of the first phase was to locate bugs in the code. Several different tasks of varying size and complexity were used for debugging.

They are of little analytical interest and will not be discussed further.

The second testing phase ran simulations of tasks that covered a wide spectrum of possible data dependency graph constructions. Although it is not possible to test every conceivable task scenario and every possible algorithm interaction, these simulations tested all of the major possibilities for the processing of tasks.

The simulation tasks contain all five of the basic algorithm execution possibilities.

(1) Only one algorithm is to be run.

(2) More than one algorithm is to be run (with none, one and several algorithms capable of being executed at any given time).

(3) The task contains branching due to decision points.

(4) The task contains loops.

(5) Some algorithms in the task have artificially high priority.

The simulations also exercised the five basic cases for the reconfiguration of the machine.

(1) No processors are idle, so DISC waits for resources to become available.

(2) The system is left in its current state.

(3) Some idle partitions are split into multiple partitions.

(4) Some idle partitions are merged into larger partitions.

(5) The system is compacted to maximize idle partition sizes.

In order to obtain enough data for analysis, at least 100 runs were performed on each of the five following tasks. These tasks are listed in Appendix C as ddg.1 through ddg.5.

(I) A task with 5 algorithms which cause all reconfigurations except compaction.

(II) The example task from Figure 5.3. This task does all machine configurings except compaction and has branching and looping.

(III) A task with only a single algorithm. It helps provide a base case for scheduling time.

(IV) A task with 9 algorithms. It contains all system reconfigurings and has algorithms with user increased priority.

(V) A task with 16 algorithms. These algorithms have no interdependencies and could be thought of as 16 separate tasks executing simultaneously.

These tasks will hereafter be referred to by the Roman numerals given above. Figures 7.1 through 7.6 show example time-resource diagrams for these tasks. These diagrams correspond to the description given in the task *tiling percentage* definition in section 7.2. Due to space constraints, the time axes in the diagrams are not shown to scale. Instead, the scheduling overhead and implementation execution time are given in the <>'s after the algorithm name. For example, '<10,200>' would indicate that 10 rules were fired to schedule the implementation and the execution time was 200 time units. The processors axes correspond to 1024 PEs. These diagrams will be discussed further in a later section.

7.3.2 Rule Firing Times

The rule firing overhead from DISC is output as the total number of rules fired in order to begin the execution of an implementation of each algorithm. By using actual rules fired instead of system time, the results are independent of the underlying machine. This method, however, presents the problem of relating number of rules fired to real

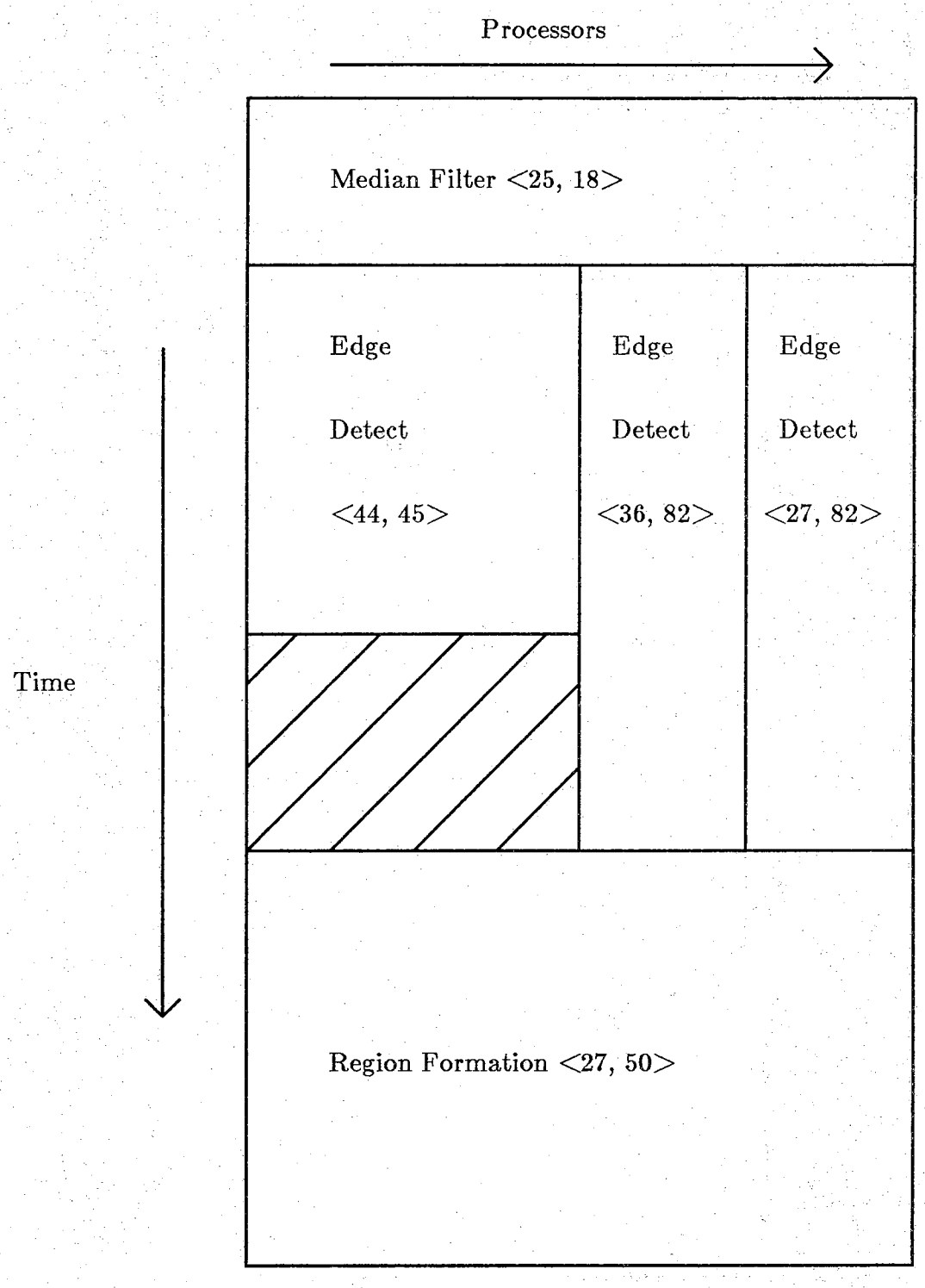


Figure 7.1 Time-Resource Diagram for Task (I)

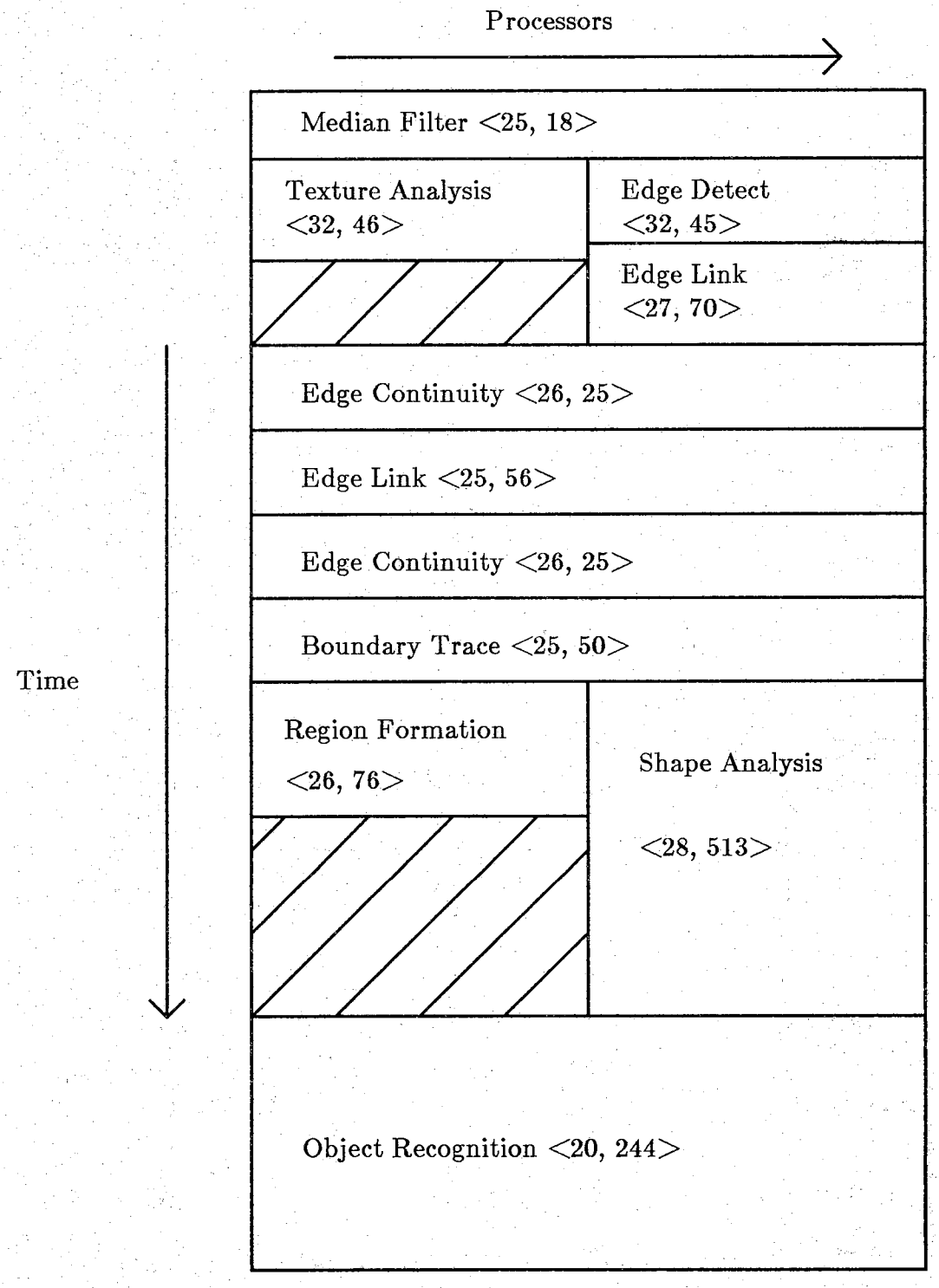


Figure 7.2 Time-Resource Diagram for Task (II)

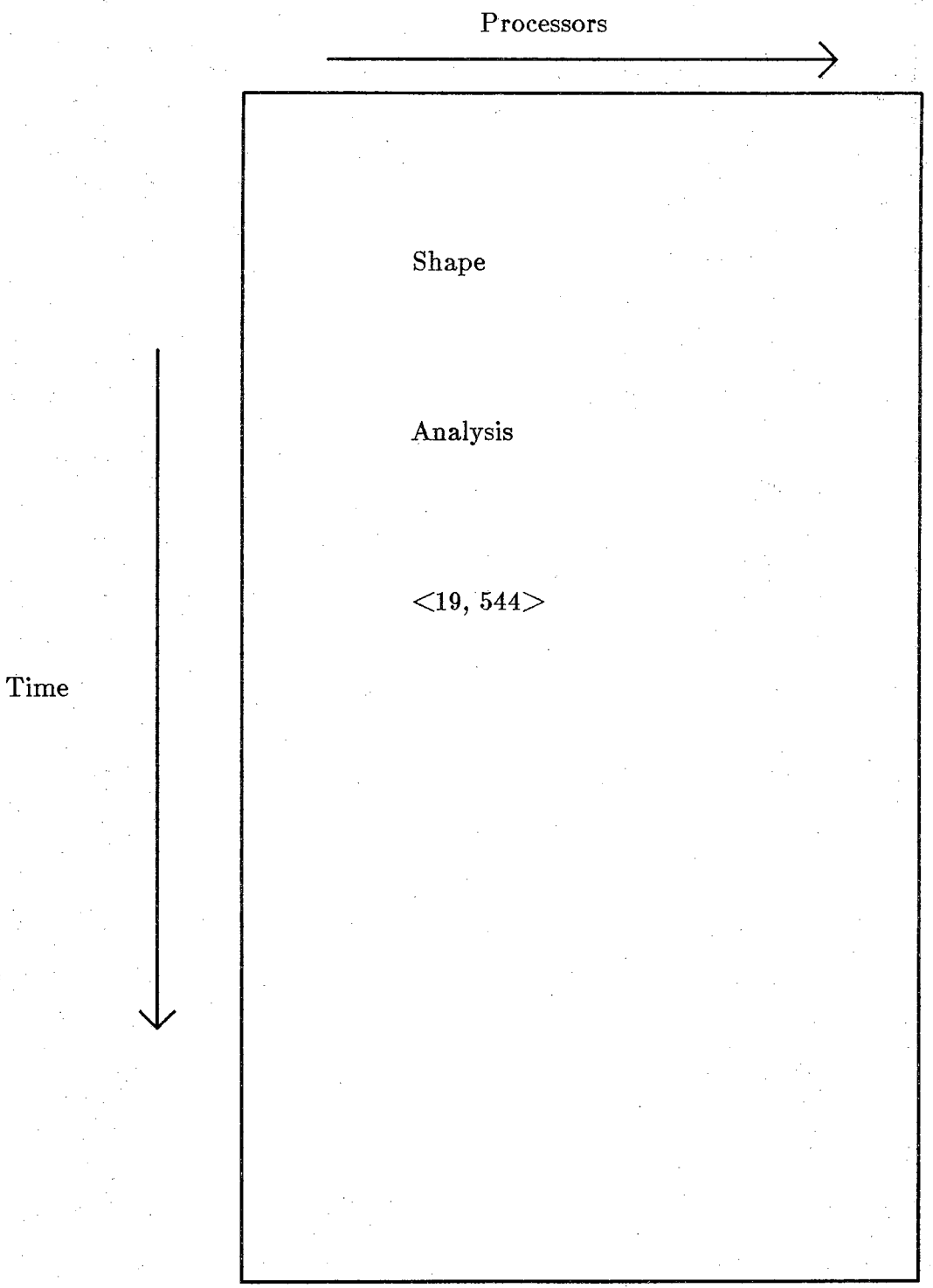


Figure 7.3 Time-Resource Diagram for Task (III)

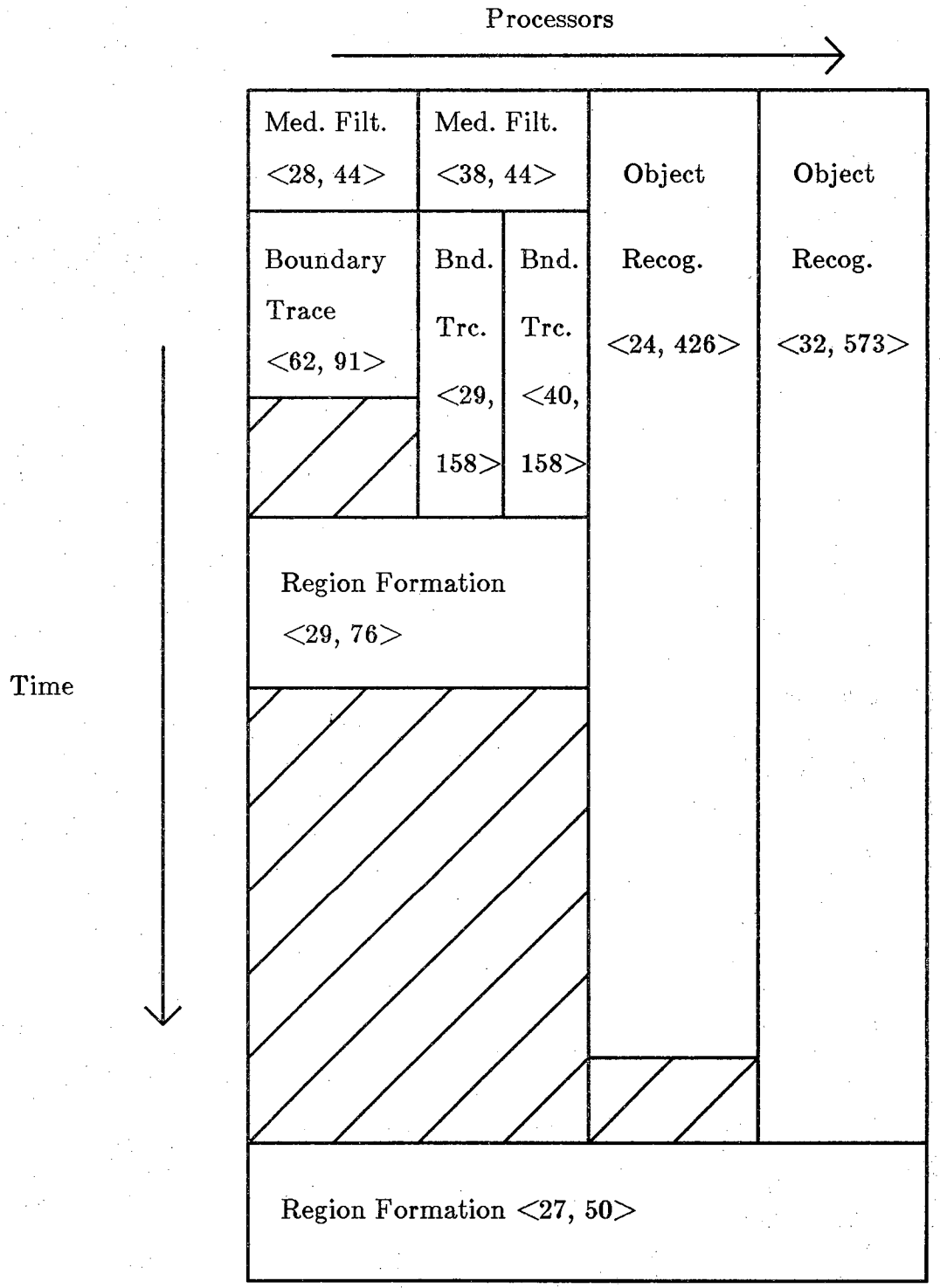


Figure 7.4 Time-Resource Diagram for Task (IV)

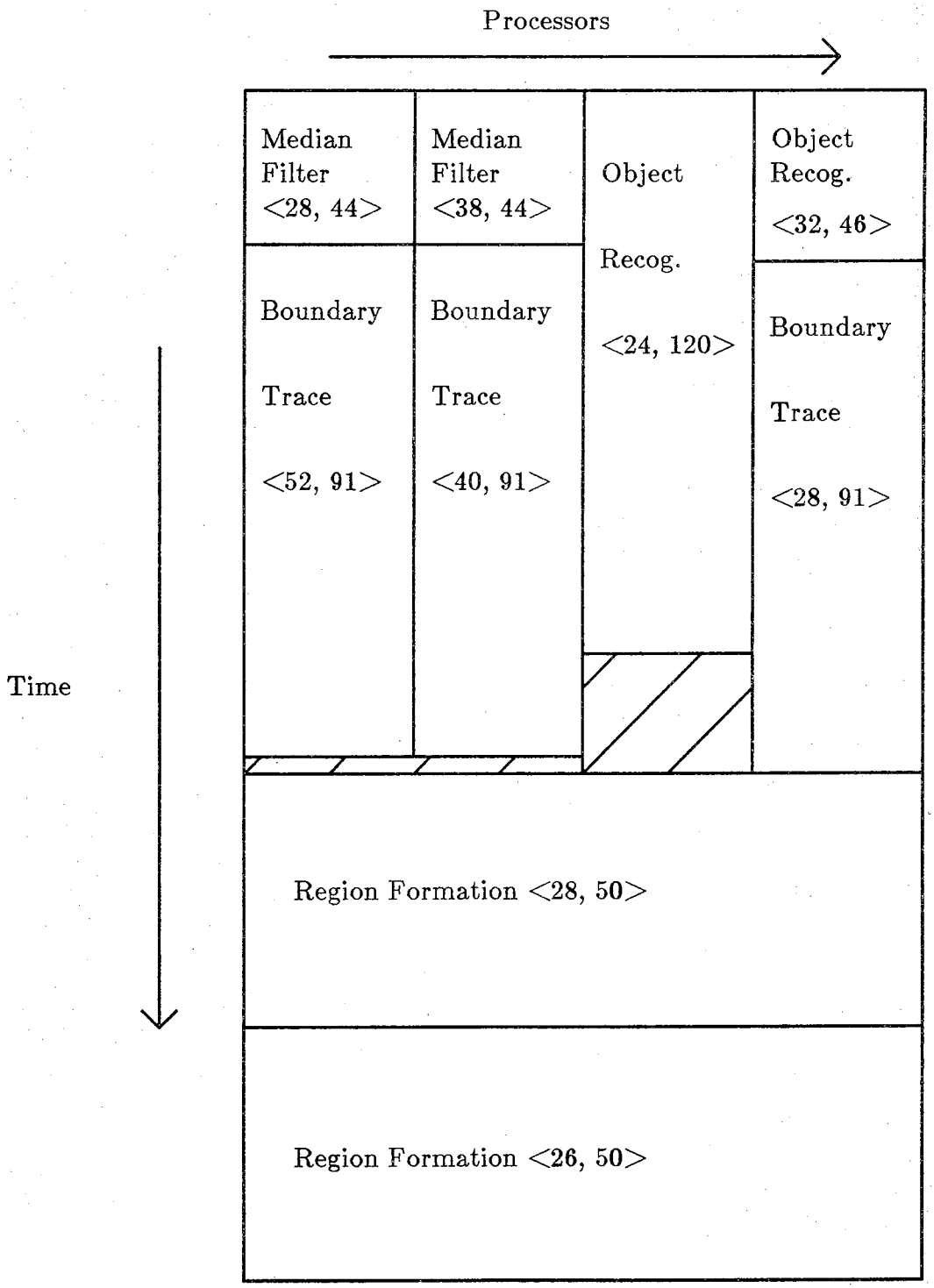


Figure 7.5 Alternate Time-Resource Diagram for Task (IV)

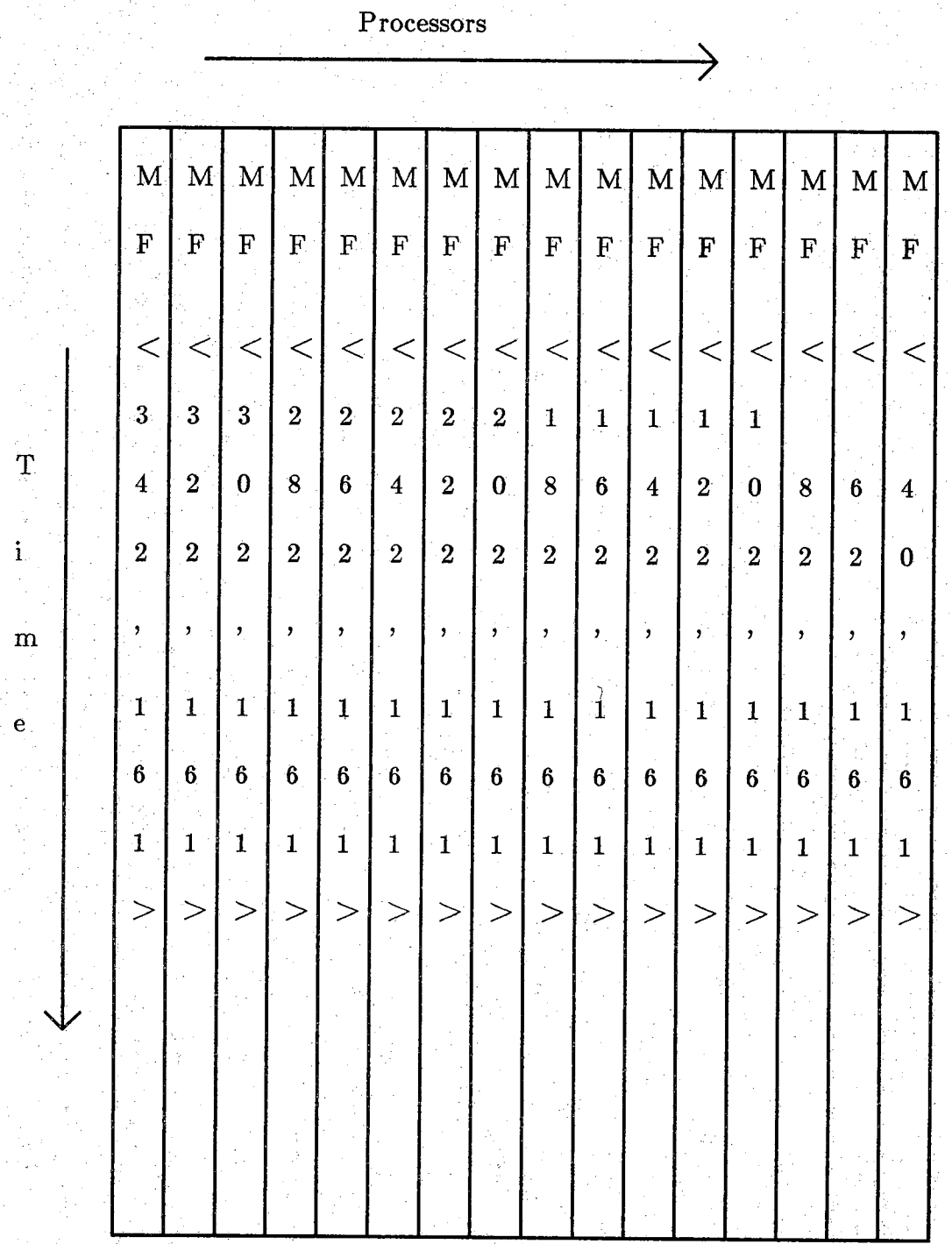


Figure 7.6 Time-Resource Diagram for Task (V)

time.

As mentioned previously, direct timing information cannot be obtained for the exact rules used by DISC. Processing added for the simulations, such as system state dumps and internal processing printouts, can increase rule firing times by several orders of magnitude. So, in order to obtain times to provide some basis for comparison, timing was done for a set of rules that contain the same number of left- and right-hand-sides as the average scheduling and reconfiguration rules in DISC. Due to the processing method of the RETE algorithm used in CLIPS [Forg82], doing timing analysis in this manner only gives an approximation. However, rules with similar numbers of LHS and RHS constructs are at least on par with each other [Forg84].

The rules used for timing purposes each have 4 LHS conditions and 3 RHS actions which are relatively complex. The rules were fired 100,000 times or more and the average rule firing time was obtained by using a system call that returns the amount of CPU time consumed by the job. These times are limited by the granularity of the internal timer (10 μ seconds) and its accuracy (it does not totally account for interrupts caused by swapping, etc.).

The tests were done with the rules both interpreted and compiled on a single-CPU VAX 11/780 and on a SUN 3/50. The results are shown in Table 7.3. Since the scheduling system could potentially be running on its own CPU, the SUN compiled times are probably the best indicators of performance. By rounding the average time for SUN compiled performance up to 20μ s (a multiple of the minimum resolution), a reasonable rule firing time estimate is obtained. By this estimate, the system is capable of firing 50,000 rules per second.

7.3.3 Simulation Results and Analysis

As was discussed in section 7.2, it is difficult to obtain performance results for entire tasks. This section will present various execution

Table 7.3 Rule Firing Times

Machine and Method	Minimum	Average	Maximum
VAX interpreted	170 μ s	510 μ s	1200 μ s
VAX compiled	$<$ 10 μ s	86 μ s	350 μ s
SUN interpreted	$<$ 10 μ s	19.8 μ s	520 μ s
SUN compiled	$<$ 10 μ s	18.8 μ s	27.2 μ s

measures in terms of number of rule firings so that an overall indication of DISC system performance can be obtained. Number of rule firings is not an absolute measure, however, since it is dependent, in part, on the number of different algorithm implementations that exist in the database. In the next section, these measures will then be employed to estimate actual performance by using the rule timings of the previous section and various simulation-time to real-time ratios.

Table 7.4 shows the overall results of the various simulations. Not all entries in the table have the same degree of accuracy or significance since different tasks contribute different amounts to each measure. Also, some measures (such as *tiling percentage*) depend heavily on the execution times of certain routines that cannot be predicted *a priori*.

Table 7.5 lists the average total execution time and total number of rules fired for each task. The total times are given in terms of arbitrary time units which will be related to real time in the next section.

The average number of rules fired per PEN algorithm is probably the best indicator of average rule firing overhead. (PEN algorithms, as was previously defined, are all those that can be executed at a given time.) This number therefore gives an average count of the amount of overhead in relation to the number of algorithms currently waiting to be run. It indicates to what extent the system is contributing to the backlog of executable algorithms and the amount of overhead needed to get all algorithms running. For example, if there are 5 PEN algorithms, an average of 113 rules will have to be fired in order to start all algorithms executing. This measure does not relate well to the user's task specification. Rather, it shows the overhead in comparison to the internal workings of the task since potential algorithm-level concurrency, the amount of machine resources, and data availability directly affect the average rule firings.

The minimum and maximum number of rules fired per PEN algorithm show the bounds for the overhead in relation to the number of algorithms currently executable. These numbers show that, if there

Table 7.4 DISC Performance Results

Measure	Result
Average number of rules fired per PEN algorithm	22.6
Minimum number of rules fired per PEN algorithm	19
Maximum number of rules fired per PEN algorithm	40
Average number of rules fired per algorithm	47.5
Scheduling Improvement	1.4
Tiling Percentage	0.77

Table 7.5 Total Run Time and Number of Rules Fired

Task	Total Run Time	Total Rules Fired
I	150	159
II	1551	281
III	509	19
IV	1171	309
V	161	3070

are 5 PEN algorithms for example, a minimum of 95 rules and a maximum of 200 rules will have to be fired in order to start the execution of all algorithms. Although these bounds are from hundreds of simulation runs and are not analytic, they do show that there are reasonable limits to the amount of overhead used by the system.

The average number of rules fired per algorithm executed relates rule firings to the actual execution of the task. It associates the system overhead with a parameter which is not affected by algorithm execution orderings, etc. This number is most useful as a comparison between very similar tasks since it does not take into account how the task is structured. It shows that a reasonable amount of overhead is incurred, on average, for the entire execution of the task. For example, a task that executes 10 algorithms including all passes through loops would only require about 475 rule firings for the total execution. The minimum and maximum firings per algorithm do not carry much

significance since they are too heavily dependent on task structure.

The measure of scheduling improvement shows how much better the execution is than if each algorithm had been run one after the other using all system resources (the worst parallel scenario). It can in general only be obtained for tasks which contain no algorithms with unpredictable execution times since unknown times cannot be reliably extrapolated to a different amount of system resources. Only tasks I, III, and V are therefore candidates for this measure. However, task III is comprised of only a single algorithm and is not of interest for this result. The scheduling improvement of 1.4 shown in Table 7.4 indicates that, on average, DISC reduced the total execution time by about 40% over the worst parallel case. The higher this number is the better, with 1.0 being the break-even case.

The tiling percentage shows the amount of time the partitions were executing algorithms instead of sitting idle. The ideal value is 1.0 which indicates no idle time. Tasks III and V have been excluded from the results for this value since their artificial structure forces zero idle time and therefore gives the ideal tiling of 1. The resulting values from the other tasks are obtained from the time-resource diagrams (T-R diagrams) shown previously. These diagrams can be used for several purposes. A graphical depiction of the packing of algorithms onto the machine can help identify weak points in the scheduling process since large areas of idle resources can easily be identified. Tiling percentage measures are obtainable directly from T-R diagrams since they are in the two-dimensional form needed for the calculations. They also provide an easy method for comparing different executions of the same task. For example, the T-R diagrams shown for task IV (Figures 7.4 and 7.5) are rather dissimilar. The difference can quickly be pinpointed as the variation in execution times for the object recognition routines.

Each task has a unique execution pattern and performance results. Although the tasks do not give an overall performance indication when they are considered individually, they do illustrate the different decision processes that the DISC system uses for scheduling. The rest of this

section discusses the performance and T-R diagrams of each task separately. For all tasks used in the simulation runs, the amount of overhead in a change from SIMD mode to MIMD mode or vice versa was considered to be negligible (which is true for the PASM system). The mode of the candidate implementation was therefore not a factor in the selection.

When the processing of task I is begun (see Figure 7.1), median filtering is the only algorithm that can be immediately executed. The processing system is therefore configured to have all 1024 processors in one partition. The two median filtering implementations in the database differ by the number of PEs required and the execution times for the given number of PEs. Both implementation meet the PE requirement, so the faster implementation is chosen. When the median filtering is finished, all three edge detection algorithms can be run. The system is therefore reconfigured into three partitions of sizes 512, 256, and 256 PEs. The specific implementations of the edge detect algorithm are chosen based on execution time since they all meet the number of PEs requirement and have the same data formats and allocations. When these algorithms finish, region formation is the only algorithm left to run so the system is reconfigured to have all PEs in a single partition. As with the other algorithms, the implementation is chosen based on the the execution time.

The tiling percentage for task I is 0.87. Processor idle time occurs in only one place in the processing. The scheduling optimality (O_τ) can be determined since every algorithm in the task has a known execution time. For this task, the best possible time is 143, giving $O_\tau = \frac{143}{150} = 0.953$. The scheduling improvement (S_τ) can also be calculated. The worst case time is 158, giving $S_\tau = \frac{158}{150} = 1.053$. These results show that DISC determined a near-optimal schedule which was an improvement over the worst case.

Task II provides a fairly complex task for analysis (see Figure 7.2). The implementation choices are basically the same as for task I with

some exceptions. When two or more algorithms are run sequentially in the same partition (such as edge detect and edge link), extra weight is given to the implementations that use the same data, format, and allocation. For algorithms such as edge link that have unknown execution times for one of the implementations, extra weight is given to the implementations which have known execution times. Whenever there are two algorithms that can be run concurrently, the system is reconfigured into two equal partitions and both are run simultaneously. The system is reconfigured into a single partition when only one algorithm can be run.

The average tiling percentage for task II is 0.768. Processor idle time occurs in two places in the task. Since the edge link algorithm has a known execution time, the idle time after the texture analysis routine always has a constant area. The idle time after the region formation algorithm varies with the execution time of the shape analysis algorithm since the object recognition algorithm cannot be run until shape analysis finishes. In general, these idle PEs account for most of the total task idle time. Since the task contains algorithms with unknown execution times, the values for O_τ and S_τ cannot be determined. However, given the relative complexity of the task, a tiling percentage of 76.8% indicates a fairly good schedule.

Task III only contains a single algorithm and is used as a boundary case (see Figure 7.3). No reconfiguration decisions need to be made except to put all PEs into a single partition if they are not already that way. The database for the shape analysis algorithm only contains one implementation, so there are no choices involved in the selection. The tiling percentage is 1.0, indicating no processor idle time. The single algorithm also forces $O_\tau = S_\tau = 1.0$ indicating that the optimal schedule is also the worst schedule.

Task IV requires more complex reconfiguration decisions than the other tasks used in testing (see Figures 7.4 and 7.5). The implementation selection decisions are similar to those for task II. DISC reconfigured the system to process algorithms concurrently

whenever possible. The two T-R diagrams shown for task IV illustrate that DISC adapts the processing of the task to the current state of the system. That is, DISC does scheduling based on the execution of the task as well as the task specification.

The average tiling percentage for task IV is 0.66. The idle time is heavily dependent on the execution times of the two object recognition algorithms. Since the task contains algorithms with unknown execution times, the values for O_τ and S_τ cannot be determined. The tiling percentage of about 66%, while not excessively small, indicates that there may be some opportunities to better assign algorithms to partitions.

Task V contains 16 algorithms that can be run simultaneously and is used as a boundary case (see Figure 7.6). Since no data dependencies exist, the faster implementation was chosen for each algorithm. The system was reconfigured to provide 16 partitions with 64 PEs each. As with task III (the other boundary case), the tiling percentage is 1.0, indicating no processor idle time. For this task, O_τ =1.0 indicating that the optimal schedule is achieved. The worst case time is 288, giving $S_\tau = \frac{288}{161} = 1.79$. The performance characteristics indicate that DISC did find the optimal schedule in this case.

Some general comments can be made about the presented results. First, several task results such as total execution time, tiling percentage, and the packing of tasks onto the machine can depend heavily on the execution times of one or more algorithms in the task. For example, consider the variation in the two T-R diagrams shown for task IV. Second, the best overall indications of DISC performance are most likely given by task II since it contains a general mix of algorithms, loops, and branching points. Finally, if the overhead time due to rule firing is small enough compared to task execution time, the numbers listed above indicate that the system can be used as a general purpose scheduling tool. This last consideration is the subject of the following section.

7.3.4 Overhead Time Estimates

By using the 20 μs rule firing time arrived at previously and the results from Table 7.4, one can see that the scheduling overhead is about 450 μs per PEN algorithm and about 950 μs per algorithm executed. Bear in mind that these times are considerably higher than could be achieved by using a dedicated processor (or parallel processor) for the execution of the DISC system.

Table 7.6 shows the actual time estimates for the scheduling overhead. Tasks I, II, and IV are most representative since they are the closest approximations of 'real' tasks. They show that a total of 3 to 6 milliseconds are used as scheduling overhead.

Table 7.6 Total Run Time and Overhead Time

Task	Total Run Time	Total Overhead Time
I	150	3.18 ms
II	1551	5.62 ms
III	509	0.38 ms
IV	1171	6.18 ms
V	161	61.4 ms

In order to get the most accurate indication of total overhead in terms of total task time, an estimate of the relation between the simulation time units and real time is needed. Since no such estimate

is available, an alternate scheme is used. The ratio of the total rule firing time per task to the total task execution time is actually the figure of most importance since relative overheads are directly comparable numbers. By multiplying the ratio of total rules fired to total simulated execution time by the ratio of the time to fire one rule to one unit of simulation time, the scheduling overhead is obtained.

Since the actual rule fire per unit simulation time ratio is unavailable, the results for various ratios are shown in Table 7.7. Tasks I, II, and IV again give the most useful results. Given the times used for the example algorithms in the simulation database, a ratio of 1000 is probably fairly realistic. By this estimate, scheduling overhead is generally less than 0.1 percent. This small percentage also means that the scheduling overhead does not affect the partition idle time calculations to any noticeable extent.

Table 7.7 Overhead Time Ratio Estimates

		Task				
		I	II	III	IV	V
Rules to Simulation Time Ratio	1	1.06	.18	.04	.26	19.1
	2	.53	.09	.02	.13	9.65
	5	.21	.04	$8*10^{-3}$	.05	3.82
	10	.11	.02	$4*10^{-3}$	.03	1.91
	50	.02	$4*10^{-3}$	$8*10^{-4}$	$5*10^{-3}$	.38
	100	.01	$2*10^{-3}$	$4*10^{-4}$	$3*10^{-3}$	.19
	1000	10^{-3}	$2*10^{-4}$	$4*10^{-5}$	$3*10^{-4}$	.02
	5000	$2*10^{-4}$	$4*10^{-5}$	$8*10^{-6}$	$5*10^{-5}$	$4*10^{-3}$

CHAPTER 8
SUMMARY

This chapter presents a summary of the research. The first section provides a brief overview of the DISC system. The second section discusses the contributions of the research. The last section discusses directions for future work in this area.

8.1 Conclusion

This thesis presents research into a system for dynamic intelligent scheduling and control of parallel processing systems. The intelligent scheduler is part of an image understanding task execution environment. One of the main functions of the environment is to isolate the user from both the details of the underlying parallel processor and the mechanics of parallel programming.

Image processing tasks are the prototype tasks used by the DISC system. Conventional scheduling methods cannot produce schedules for most tasks of this type. A dynamic controller, however, is not bound by this limitation and can reconfigure the machine and process subtasks based on the current state of the parallel processing system.

The DISC system uses the execution characteristics of image processing routines, rule-based heuristics, and the current system state to produce and continually update a schedule for the subtasks that comprise the task. The scheduling system attempts to achieve decreased execution time by balancing the overall processing scenario of

the task with the needs of the individual routines that make up the task.

By using the DISC system, the efficient and rapid prototyping of image understanding tasks becomes possible. It allows task specifications that are independent of the underlying parallel processing system. Use of the DISC system requires no knowledge of parallel programming or the underlying parallel architecture.

8.2 Contributions

The major contributions from this research are in the area of operating system design for parallel processing systems. The research presents a method of scheduling tasks which have unknown execution characteristics. This system therefore fills a gap left by conventional scheduling methods.

The DISC system allows a user with little or no knowledge of parallel processing software or hardware to use a parallel processing system. The scheduling and reconfiguration heuristics which were formulated for DISC are applicable to any reconfigurable architecture, so the system can be easily ported to many parallel processors. Algorithms and heuristics for partition merging and splitting and machine compaction were devised. The heuristics do not apply exclusively to image processing. Any tasks that can be split into well-defined subtasks can be processed with the DISC system as long as the appropriate algorithm database exists.

A set of characteristics for gauging task execution performance were presented. These performance criteria (average time per algorithm, tiling percentage, task execution time, schedule optimality, and schedule improvement), characterize a range of aspects of the overall execution of a task. They provide a method for the comparison of different scheduling strategies as well as a method for gauging the performance of an individual scheduling strategy. These features

represent an extension of and complement to the characteristics conventionally used to classify parallel algorithm performance.

In the area of parallel processing, the research has presented a method of generating task descriptions that are directly portable to any parallel architecture running the DISC system. The system also provides a mechanism for the rapid and efficient prototyping of tasks.

A task language was defined in terms of a data dependency graph and a parser was developed for the structure. The concept of a reduced data dependency graph was introduced to minimize the amount of redundant information present in the task graph.

The DISC system prototype was developed and its performance was evaluated. Testing was done on a number of tasks that exercised different aspects of the scheduling strategy. The results show that the DISC system adds little overhead to the task and that the schedules determined by DISC are a definite improvement over worst-case methods. The minimal overhead caused by DISC indicate that it could be used as a general purpose scheduling tool as well as a rapid prototyping system.

8.3 Future Work

There are two areas for future extensions to this research. The first area is improving DISC's performance. The second area is expanding its capabilities.

One method of improving the performance of DISC is to improve the heuristics used. The improvement could come either from modifying current heuristics to better handle the tasks or from adding new heuristics to handle more exceptions to the basic processing scheme. The best way to derive new heuristics would be to run many realistic tasks and try to isolate the locations where DISC is making choices that could be improved. These areas are prime candidates for adding heuristics to account for processing scenarios that do not

conform well to the expected operation.

There are several possibilities for expanding the capabilities of the DISC system. Probably the most useful would be to change the DISC from a simulation configuration to a working system. This process would involve removing the performance monitors and providing the actual low-level operating system interface.

The other area for expansion is to allow types of tasks other than image processing to execute on the system. Possibilities that are immediately apparent are computer graphics, speech processing, and signal processing. Further work needs to be done to target processing areas that have well-defined algorithmic primitives and can be analyzed for inclusion in the algorithm database.

LIST OF REFERENCES

LIST OF REFERENCES

[Adam82] G. B. Adams III and H. J. Siegel, "The Extra Stage Cube: a Fault-Tolerant Interconnection Network for Supersystems," *IEEE Transactions on Computers,* VOL C-31, May 1982, pp. 443-454.

[Aho79] A. V. Aho, J. D. Ullman, and M. Yannakakis, "Modeling Communication Protocols by Automata," *20th Symposium on the Foundations of Computer Science,* Oct. 1979, pp. 267-273.

[Aho86] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools,* Addison_Wesley, 1986.

[Alle83] J. R. Allen, "Dependence Analysis for Subscript Variables and Its Application to Program Transformations," *PhD. Thesis, Rice University,* Apr. 1983.

[Ande85] V. S. Andersen, T. Haugland, and O. Sorasen, "CESAR - A Programmable Systolic Array Multiprocessor System," *1st International Conference on Supercomputing Systems,* Dec. 1985, pp. 8-15.

[Babb84] R. G. Babb II, "Parallel Processing with Large-Grain Data Flow Techniques," *IEEE Computer,* VOL 17, Jul. 1984, pp. 55-61.

[Barn68] G. H. Barnes, R. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The ILLIAC IV Computer," *IEEE Transactions on Computers,* VOL C-17, Aug. 1968, pp. 746-757.

[Batc74] K. E. Batcher, "STARAN Parallel Processor System Hardware," *AFIPS 1974 NCC,* May 1974, pp. 405-410.

[Batc80] K. E. Batcher, "Design of a Massively Parallel Processor," *IEEE Transactions on Computers,* VOL C-29, Sep. 1980, pp. 836-840.

[Batc82] K. E. Batcher, "Bit Serial Parallel Processing Systems," *IEEE Transactions on Computers,* VOL C-31, May 1982, pp. 377-384.

[Beet85] J. Beetem, M. Denneau, and D. Weingarten, "The GF11 Supercomputer," *12th Annual International Symposium on Computer Architecture,* Jun. 1985, pp. 108-115.

[Bouk72] W. J. Bouknight, S. A. Denenberg, D. E. McIntryre, J. M. Randall, A. H. Sameh, and D. L. Slotnick, "The ILLIAC IV System," *Proceedings of the IEEE,* VOL 60, Apr. 1972, pp. 369-388.

[Bran83] D. Brand and P. Zafiropulo, "On Communicating Finite-State Machines," *JACM, VOL 30, No. 2, Apr. 1983, pp. 323-342.*

[Brin75] P. Brinch-Hansen, "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering,* VOL SE-1, No. 2, Jun. 1975, pp. 199-207.

[Broc79] J. D. Brock and L. B. Monty, "Translation and Optimization of Data Flow Programs," *Proceedings of the 1979 International Conference on Parallel Processing,* Aug. 1979, pp. 46-54.

[Bron82] E. C. Bronson and L. J. Siegel, "A Parallel Architecture for Acoustic Processing In Speech Understanding," *1982 International Conference on Parallel Processing,* Aug. 1982, pp. 307-312.

[Burt87] P. J. Burt and G. S. van der Wal, "Iconic Image Analysis with the Pyramid Vision Machine (PVM)," *Proceedings of the 1987 IEEE Workshop on Computer Architecture for Pattern Analysis and Machine Intelligence,"* Oct. 1987, pp. 137-144.

[Cann86] J. F. Canny, "A Computational Approach to Edge Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* VOL PAMI-8, Nov. 1986, pp. 679-698.

[Casa87] T. L. Casavant, H. G. Dietz, T. Schwederski, P. C-Y. Sheu, and H. J. Siegel, "Software Plans for PASM," *Proceedings of the 2nd International Conference on Supercomputing,* May 1987.

[Chu87] C. H. Chu, E. J. Delp, L. H. Jamieson, H. J. Siegel, F. J. Weil, and A. B. Whinston, "A Model for an Intelligent Operating System for Executing Tasks on a Reconfigurable Parallel Architecture," *Supercomputing Research Center Technical Report Series,* SRC-TR-87-007, Lanham, MA, Nov. 1987.

[Clar84] K. Clark and S. Gregory, "PARLOG: Parallel Programming in Logic," *Research Report DOC 84/4,* Department of Computing, Imperial College of Science and Technology, Apr. 1984.

[Cler87] Ph. Clermont and A. Merigot, "Real Time Synchronization in a Multi-SIMD Massively Parallel Machine," *Proceedings of the 1987 IEEE Workshop on Computer Architecture for Pattern Analysis and Machine Intelligence,*" Oct. 1987, pp. 131-136.

[Cone81] J. S. Conery and D. F. Kibler, "Parallel Interpretation of Logic Programs," *Journal of the ACM* May 1981, pp. 163-170.

[Cone84] J. S. Conery and D. F. Kibler, "AND Parallelism in Logic Programs," in *Readings in Parallel Logic Programming,* IEEE Tutorial on Logic Programming and Parallel Processing, D. DeGroot, ed., Jul. 1984, pp. 13-17.

[Crow85] W. Crowther, J. Goodhue, R. Thomas, W. Milliken, and T. Blackadar, "Performance Measures on a 128-Node Butterfly Parallel Processor," *1985 International Conference on Parallel Processing,* Aug. 1985, pp. 531-540.

[Culb88] C. Culbert, *CLIPS Reference Manual, Version 4.2 of CLIPS,* Artificial Intelligence Section, Lyndon B. Johnson Space Center, NASA, Apr. 1988.

[Davi82] A. L. Davis and R. M. Keller, "Data Flow Program Graphs," *IEEE Computer,* Feb. 1982, pp. 26-41.

[Davi87] S. Davis, ed., *Genetic Algorithms and Simulated Annealing,* Pitman Publishing, 1987.

[DeGr84] D. DeGroot, "Restricted AND-Parallelism," *Proceedings of the 1984 International Conference on Fifth Generation Computers,* Nov. 1984.

[Delp85] E. J. Delp, H. J. Siegel, A. B. Whinston, and L. H. Jamieson, "An Intelligent Operating System for Executing Image Understanding Tasks on a Reconfigurable Parallel Architecture," *Proceeding of the 1985 IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management,* Nov. 1985, pp. 217-224.

[Diet85] H. Dietz and D. Klappholz, "Refined C: A Sequential Language for Parallel Programming," *Proceedings of the 1985 International Conference on Parallel Processing,* Aug. 1985, pp. 442-449.

[Diet86] H. Dietz and D. Klappholz, "Refined FORTRAN: Another Sequential Language for Parallel Programming," *Proceeding of the 1986 International Conference on Parallel Processing,* Aug. 1986, pp. 184-191.

[Dixi87] V. Dixit and D. J. Moldovan, "Semantic Network Array Processor and Its Applications to Image Understanding," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* VOL PAMI-9, Jan. 1987, pp. 153-160.

[Finn85] D. J. Finn, "Simulation of the Hughes Data Flow Multiprocessor Architecture," *1st International Conference on Supercomputing Systems,* Dec. 1985, pp. 331-340.

[Flyn66] M. J. Flynn, "Very High-Speed Computing Systems," *Proceedings of the IEEE,* VOL 54, Dec. 1966, pp. 1901-1909.

[Forg82] C. L. Forgy, "Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem," *Artificial Intelligence, VOL 19, 1982, pp. 17-37.*

[Forg84] C. Forgy, A. Goopta, A. Newell, and R. Wedig, "Initial Assessment of Architectures for Production Systems," *Proceedings of the National Conference on Artificial Intelligence,* Aug. 1984, pp. 116-120.

[Gajs82] D. D. Gajski, D. A. Padua, D. J. Kuck, R. H. Kuhn, "A Second Opinion on Data Flow Machines and Languages," *IEEE Computer,* VOL 15, Feb. 1982, pp. 58-69.

[Giar88] J. C. Giarratano, *CLIPS User's Guide, Version 4.2 of CLIPS,* Artificial Intelligence Section, Lyndon B. Johnson Space Center, NASA, Jun. 1988.

[Gott83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared-Memory Parallel Computer," *IEEE Transactions on Computers,* VOL C-32, Feb. 1983, pp. 175-189.

[Hill85] W. D. Hillis, *The Connection Machine,* MIT Press, Cambridge, MA, 1985.

[Holl59] J. H. Holland, "A Universal Computer Capable of Executing an Arbitrary Number of Sub-programs Simultaneously," *Proceedings of the EJCC,* 1959, pp. 108-113.

[Horo78] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms,* Computer Science Press, 1978.

[Houg87] A. A. Hough and J. E. Cuny, "Belvedere: Prototype of a Pattern-Oriented Debugger for Highly Parallel Computation," *Proceedings of the 1987 International Conference on Parallel Processing,* Aug. 1987, pp. 735-738.

[Huec71] M. H. Hueckel, "An Operator Which Locates Edges in Digitized Pictures," *Journal of the ACM,* VOL 18, 1971, pp. 113-125.

[Huec73] M. H. Hueckel, "A Local Visual Edge Locator Which Recognizes Edges and Lines," *Journal of the ACM,* VOL 20, 1973, pp. 634-647.

[Huma86] Human Devices, Parallon™ *Parallel Processor,* Advertisement

[Hwan84] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing,* McGraw-Hill, 1984.

[INMO83] INMOS Ltd., *occam Programming Manual,* Bristol, 1983.

[Jami86] L. H. Jamieson, P. T. Mueller Jr., and H. J. Siegel, "FFT Algorithms for SIMD Parallel Processing Systems," *Journal of Parallel and Distributed Computing,* VOL 3, 1986, pp. 48-71.

[Jami87] *The Characteristics of Parallel Algorithms,* L. H. Jamieson, D. B. Gannon, and R. J. Douglass, eds., MIT Press, 1987.

[Jeng88] M. Jeng and H. J. Siegel, "Dynamic Partitioning in a Class of Parallel Systems," *Proceedings of the 8th International Conference on Distributed Computing Systems,*" Jun. 1988, pp. 33-40.

[Jone77] A. K. Jones, R. J. Chansler Jr., I. Durham, P. Feiler, and K. Schwans, "Software Management of Cm* - a Distributed Multiprocessor," *AFIPS 1977 NCC,* Jun. 1977, pp. 657-663.

[Kasc80] M. J. Kascic Jr., "Vector Processing, Problem or Opportunity?," *IEEE 1980 Compcon,* Feb. 1980, pp. 270-276.

[Kell80] R. M. Keller, G. Lindstrom, and S. Patil, "Data-Flow Concepts for Hardware Design," *IEEE 1980 Compcon,* Feb. 1980, pp. 105-111.

[Kram84] J. Krammer, J. Magee, M. Sloman, K. P. Twidle, and N. Dulay, "The CONIC Programming Language: Version 2.4," *Research Report DOC 84/19,* Imperial College, Oct. 1984.

[Kuck74] D. Kuck, P. Budnik, S-C. Chen, E. Davis Jr., J. Han, P. Kraska, D. Lawrie, Y. Muraoka, R. Strebendt, and R. Towle, "Measurements of Parallelism in Ordinary FORTRAN Programs," *Computer,* VOL 7, No. 1, Jan. 1974, pp. 37-46.

[Kuck76] D. J. Kuck, "Parallel Processing of Ordinary Programs," in *Advances in Computers,* M. Rubinoff and M. C. Yovits, eds., Academic Press, 1976, pp. 119-179.

[Kuck80] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors," *Fourth International Computer Software and Applications Conference,* Oct. 1980.

[Kueh85] J. T. Kuehn, T. Schwederski, and H. J. Siegel, "Design of a 1024-Processor PASM System," *1st International Conference on Supercomputing Systems,* Dec. 1985, pp. 603-612.

[Kueh85b] J. T. Kuehn and H. J. Siegel, "Extensions to the C Programming LAnguage for SIMD/MIMD Parallelism," *Proceedings of the 1985 International Conference on Parallel Processing,* Aug. 1985, pp. 232-235.

[Kung82] S. Y. Kung, R. J. Gal-Ezeri, and K. S. Arun, "Wavefront Array Processor: Architecture, Language, and Applications," *1982 Conference on Advanced Research in VLSI, MIT* Jan. 1982, pp. 4-19.

[Kung82b] H. T. Kung, "Why Systolic Architectures?," *IEEE Computer,* VOL 15, No. 1, Jan. 1982, pp. 37-46.

[Lee80] R. B. Lee, "Empirical Results on the Speed, Efficiency, Redundancy, and Quality of Parallel Computations," *Proceedings of the 1980 International Conference on Parallel Processing,* 1980, pp. 91-100.

[McGr85] J. R. McGraw, et. al., "SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual," *Report M-146, Revision 1,* Lawrence Livermore National Laboratory, Mar. 1985.

[Mehr85] P. Mehrotra and J Van Rosendale, "The BLAZE Language: A Parallel Language for Scientific Programming," *ICASE Report No. 85-29,* NASA Langley Research Center, May 1985.

[Mode76] H. S. Modell and T. M. Sparr, "Coordination of Parallel Processes in PL/1," *Proceedings of the 1976 International Conference on Parallel Processing,* Aug. 1976, pp. 247-253.

[Moor85] T. P. Moore and A. J. de Geus, "Simulated Annealing Controlled by a Rule-Based Expert System," *1985 IEEE International Conference on Computer-Aided Design (ICCAD '85),* Nov. 1985, pp. 200-202.

[Mudg82] T. N. Mudge, E. J. Delp, L. J. Siegel, and H. J. Siegel, "Image Coding Using the Multiprocessor System PASM," *Proceedings of the 1982 IEEE Computer Society Conference on Pattern Recognition and Image Processing,* Jun. 1982, pp. 200-205.

[Nutt77] G. J. Nutt, "Microprocessor Implementation of a Parallel Processor," *4th Annual Symposium on Computer Architecture,* Mar. 1977, pp. 147-152.

[Nutt77b] G. J. Nutt, "A Parallel Processor Operating System Comparison," *IEEE Transactions on Software Engineering,* VOL SE-3, Nov. 1977, pp. 467-475.

[Poly86] C. D. Polychronopoulos and U. Banerjee, "Speedup Bounds and Processor Allocation for Parallel Programs on Multiprocessors," *Proceedings of the 1986 International Conference on Parallel Processing,* Aug. 1986, pp. 961-968.

[Prew70] J. M. S. Prewitt, "Object Enhancement and Extraction," in *Picture Processing and Psychopictorics,* B. S. Lipkin and A. Rosenfeld, eds., Academic Press, 1970, pp. 75-149.

[Reev80] A. P. Reeves, J. D. Bruner, and M. S. Poret, "The Programming Language Parallel Pascal," *Proceedings of the 1980 International Conference on Parallel Processing,* Aug. 1980, pp. 5-6.

[Rice85] T. A. Rice and L. H. Jamieson, "Parallel LISP," *Technical Report TR-EE 85-2,* School of Electrical Engineering, Purdue University, Jan. 1985.

[Rice85b] T. A. Rice and L. H. Jamieson, "Parallel Processing for Computer Vision," in *Integrated Technology for Parallel Image Processing,* S. Levialdi, ed., Academic Press, 1985.

[Rinn76] A. H. G. Rinnooy Kan, *Machine Scheduling Problems,* Martinus Nijhoff, 1976.

[Robe65] L. G. Roberts, "Machine Perception of Three-Dimensional Solids," in *Optical and Electro-Optical Information Processing,* J. T. Tippet, ed., MIT Press, 1965, pp. 159-197.

[Schw87] T. Schwederski, "The PASM Parallel Processing System: Hardware Design and Operating System Concepts," *Ph.D. Thesis, Purdue University,* Dec. 1987.

[Schw88] T. Schwederski, H. J. Siegel, and T. L. Casavant, "A Model of Task Migration in Partitionable Parallel Processing Systems," *Frontiers '88: 2nd Symposium on the Frontiers of Massively Parallel Computation,* 1988.

[Seit85] C. L. Seitz, "The Cosmic Cube," *Communications of the ACM,* Jan. 1985, pp. 22-23.

[Sejn80] M. C. Sejnowski, E. T. Upchurch, R. N. Kapur, D. P. S. Charlu, and G. J. Lipovski, "An Overview of the Texas Reconfigurable Array Computer," *AFIPS 1980 NCC,* Jun 1980, pp. 631-641.

[Shap77] H. D. Shapiro, "Comparison of Various Methods of Detecting and Utilizing Parallelism in a Single Instruction Stream," *Proceedings of the 1977 International Conference on Parallel Processing,* Aug. 9177, pp. 67-76.

[Shap83] E. Y. Shapiro, "A Subset of Concurrent Prolog and Its Interpreter," *Technical Report TR-003,* ICOT - Institute for New Generation Computer Technology, Feb. 1983.

[Sieg80] H. J. Siegel, "The Theory Underlying the Partitioning of Permutation Networks," *IEEE Transactions on Computers,*" VOL C-29, No. 9, Sep. 1980, pp. 791-801.

[Sieg81] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Meuller Jr., H. E. Smalley Jr., and S. D. Smith, "PASM: a Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition," *IEEE Transactions on Computers,* VOL C-30, Dec. 1981, pp. 934-947.

[Sieg81b] L. J. Siegel, "Image Processing on a Partitionable SIMD Machine," in *Languages and Architectures for Image Processing* M. J. B. Duff and S. Levialdi, eds., Academic Press, 1981, pp. 292-300.

[Sieg82] L. J. Siegel, H. J. Siegel, and P. H. Swain, "Performance Measures for Evaluating Algorithms for SIMD Machines," *IEEE Transactions on Software Engineering,* VOL SE-8, No. 4, Jul. 1982, pp. 319-331.

[Slee84] M. R. Sleep and J. R. Kennaway, "The Zero Assignment Parallel Processor (ZAPP) Project," in *Distributed Computing Systems Programme,* D. A. Duce, ed., Peter Peregrinus Ltd., 1984, pp. 250-269.

[Slot62] D. L. Slotnick, W. C. Borch, and R. C. McReynolds, "The Solomon Computer - a Preliminary Report," *Proceedings of the 1962 Workshop on Computer Organization,* Washington, D.C., pp. 66-92.

[Smit84] K. D. Smith and L. H. Jamieson, "Parallel Computation of Normalized Fourier Descriptors," *Proceedings of the Twenty-second Annual Allerton Conference on Communication, Control, and Computing,* Oct. 1984.

[Swan77] R. J. Swan, S. Fuller, and D. P. Siewiorek, "Cm*: a Modular Multimicroprocessor," *AFIPS 1977 NCC,* Jun. 1977, pp. 637-644.

[Swan77b] R. J. Swan, A. Bechtolsheim, K. W. Lai, and J. K. Ousterhout, "The Implementation of the Cm* Multimicroprocessor," *AFIPS 1977 NCC,* Jun. 1977, pp. 645-655.

[Toma67] R. M. Tomasulo, "An Efficient Algorithm for Automatic Exploitation of Multiple Execution Units," *IBM Journal of Research and Development,* Jan. 1967, pp. 25-33.

[Tsao82] Y-F. Tsao and K. S. Fu, "A 3D Parallel Skeletonwise Thinning Algorithm," *Proceedings of the 1982 IEEE Computer Society Conference on Pattern Recognition and Image Processing,* Jun. 1982, pp. 678-683.

[Tuom81] D. L. Tuomenoksa and H. J. Siegel, "Applications Of Two-dimensional Bin Packing to Task Scheduling in PASM," *Allerton Conference on Communication, Control, and Computing,* Oct. 1981, p. 542.

[Unge58] S. H. Unger, "A Computer Oriented Toward Spatial Problems," *Proceedings of the IRE,* Oct. 1958, pp. 17-44.

[vanE84] M. H. van Emden and G. J. de Lucena Filho, "Predicate Logic as a Language for Parallel Programming," in *Readings in Parallel Logic Programming,* IEEE Tutorial on Logic Programming and Parallel Processing, D. DeGroot, ed., Jul 1984, pp. 134-139.

[vanL87] P. J. M. van Laarhoven and E. H. l. Aarts, *Simulated Annealing: Theory and Applications,* D. Reidel, Boston, 1987.

[Warp78] M. R. Warpenburg and L. J. Siegel, "SIMD Image Resampling," *IEEE Transactions on Computers,* VOL C-31, NO. 10, Oct. 1982, pp. 934-942.

[Walt78] D. L. Waltz, "A Parallel Model for Low-Level Vision," in *Computer Vision Systems,* A. R. Hanson and E. M. Riseman, eds., Academic Press, 1978, pp. 175-186.

[Weil87] F. J. Weil, L. H. Jamieson, and E. J. Delp, "Some Aspects of an Image Understanding Database for an Intelligent Operating System," *Proceedings of the 1987 Workshop on Computer Architecture for Pattern Analysis and Machine Intelligence,* Oct. 1987, pp. 203-208.

[Weil88] F. J. Weil, L. H. Jamieson, and E. J. Delp, "An Algorithm Database for an Image Understanding Task Execution Environment," in *Multicomputer Vision,* S. Levialdi, ed., Academic Press, 1988, pp. 35-51.

[Well71] M. B. Wells, *Elements of Combinatorial Computing,* Pergamon Press, 1971.

[Widd80] L. C. Widdoes Jr., "The S-1 Project: Developing High-Performance Digital Computers," *IEEE 1980 Compcon,* Feb. 1980, pp. 282-291.

APPENDICES

Appendix A
The Machine Partitioning Problem

As was previously mentioned, possible machine configurations grow at least exponentially with the number of PEs in the machine and the number of desired partitions. This appendix presents a mathematical treatment of the partitioning problem.

The discussion of repartitioning presented in this appendix pertains to the type of interconnection network and control hierarchy used in the PASM parallel processor. The ICN is a hypercube configuration. There is an added restriction that micro-controllers grouped into a partition of size 2^p must have the same low-order 10—p bits in their physical address. Partition sizes must therefore be a power of 2. This restriction impresses a binary tree structure to the machine. The partitioning of an unrestricted hypercube is actually a much more complex problem and will not be discussed here.

The full PASM system has 32 MCs and the graph for the system has 63 nodes. For the sake of simplicity, the example system discussed here will have only 8 MCs (and a total of 15 nodes). Figure A.1 shows the binary tree for the 8 MC system. The MCs in the system are represented by the leaf nodes in the tree (labeled 0 through 7). The non-leaf nodes represent possible partitions in the machine and exist at the conceptual level only. Let G_M denote the binary tree that represents the control structure of a given machine M.

The binary tree structure of a machine partitioning can be stated as follows: A set of MCs can be in the same partition iff two conditions hold.

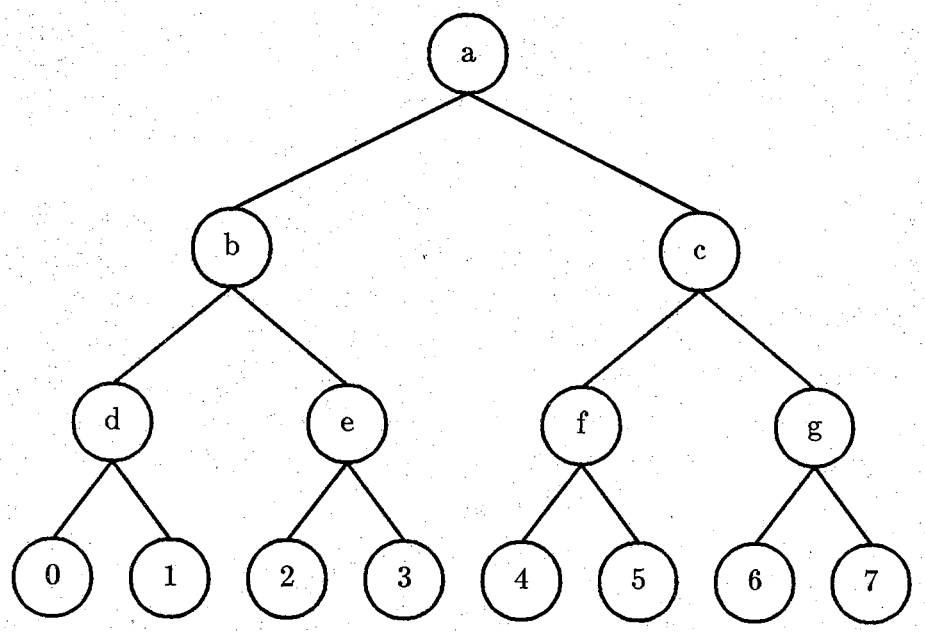


Figure A.1 An 8 PE Machine Representation

(1) All MCs in the partition are in the same subtree S_{G_M} of G_M.

(2) *All* leaf nodes of S_{G_M} are in the partition.

These conditions force partition sizes to be a power of 2 and supply some regularity to the partitioning process. That is, to obtain a partition of size C in a G_M with N leaf nodes, the only possibilities are the subtrees with root nodes at depth logN − logC + 1. (All logarithms in this appendix are base 2). For the example tree in Figure A.1, if a partition of size 1 is desired, then the root of the subtree is at depth

$$\log 8 - \log 1 + 1 = 3 - 0 + 1 = 4$$

which is the depth of the leaf nodes as expected. Partitions of size 4 are created at depth

$$\log 8 - \log 4 + 1 = 3 - 2 + 1 = 2$$

Due to condition (2) above, any partition of the machine can be represented by the root of its subtree. In the example tree of Figure A.1, node *b* represents the partition (0 1 2 3), node *g* represents the partition (6 7) and node *a* represents the whole machine in one partition (0 1 2 3 4 5 6 7). When a node is to represent a partition in this way, it is shown as a leaf node.

The number of ways to create P partitions on a machine M can be defined in the following way.

Given a full, labeled binary tree G_M with N leaf nodes, determine the number of connected subtrees of G_M that satisfy the following conditions:

(1) All nodes of a subtree have either 0 or 2 children.

(2) All subtrees must contain the root of G.

(3) Given some number $P \leqq N$, there are exactly P leaf nodes in the subtree.

Although not all sources agree on the exact meaning of the term, *binary tree* will be used throughout this appendix for trees that meet condition (1) above. Figure A.2 illustrates the possible machine partitionings for a 4 MC machine when 3 partitions are desired. Figure A.2.A shows the binary tree for the whole machine. Figure A.2.B and A.2.C show the only two possible machine configurations for 3 partitions. They correspond to the partitionings ((0) (1) (2 3)) and ((0 1) (2) (3)) respectively.

A recurrence relation can be derived that will give the number of partitionings possible for a given machine size and number of partitions. Let $\Phi(N,P)$ be the number of subtrees of G_M that meet the above conditions with N being the number of MCs in the machine and P being the number of desired partitions. Theorem 1 states the recurrence relation.

Theorem 1

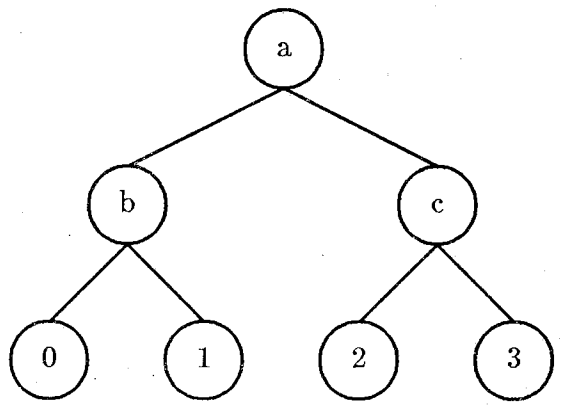
$$\Phi(N,P) = \sum_{i=1}^{P-1} \Phi(\frac{N}{2},i)\Phi(\frac{N}{2},P-i)$$

or, letting n=logN,

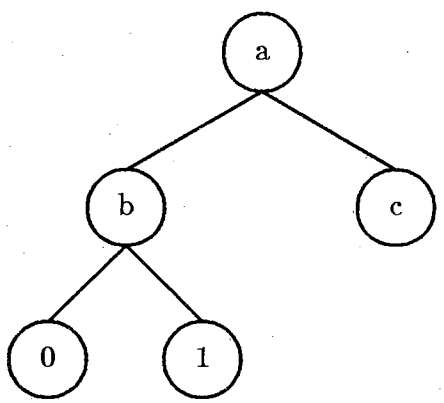
$$\Phi(n,P) = \sum_{i=1}^{P-1} \Phi(n-1,i)\Phi(n-1,P-i)$$

Proof:

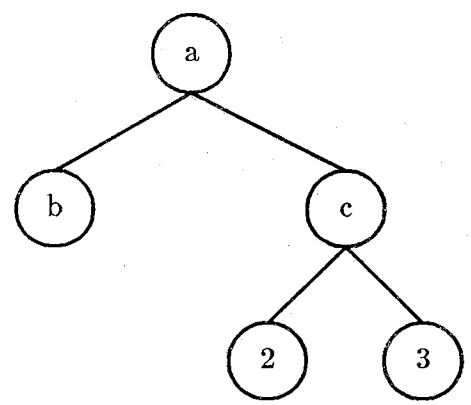
Since each subtree representing a partitioning must contain the root of G_M, be connected, and each node has either 0 or 2



A



B



C

Figure A.2 3 Partitionings of a 4 MC Machine

children, we can get P leaf nodes by putting 1 leaf node in the left half of the tree and P−1 leaf nodes in the right half, or 2 leaf nodes in the left half and P−2 leaf nodes in the right half, ..., or P−1 leaf nodes in the left half and 1 in the right half. The number of ways to put i leaf nodes in one half of the tree is $\Phi(\frac{N}{2},i)$ since each half of the tree has $\frac{N}{2}$ leaf nodes. Therefore, there are $\Phi(\frac{N}{2},i)\Phi(\frac{N}{2},P-i)$ ways to make P partitions by putting i in one half of the tree and P−i in the other half. The total number of ways to make P partitions is the sum over all value of i less than P.

■

Given that $\Phi(N,P)$ is a nonlinear, 2-variable recurrence, it is not trivial to find a closed-form solution. It would seem that the way to attack this problem is to consider it as a subtree construction problem. However, $\Phi(N,P)$ is a useful equation for generating values. Table A.1 lists the values of $\Phi(N,P)$ for values of N up to 32. Figures A.3, A.4, and A.5 show values of $\Phi(N,P)$ for N=8, N=16, and N=32 respectively. Note that the values of $\Phi(N,P)$ increase very quickly with both P and N.

Let $\Psi(N)$ be the maximum of $\Phi(N,P)$ over all P. That is, $\Psi(N)$ is the maximum number of ways a machine of size N can be partitioned. It can be shown that these values increase at least exponentially.

Theorem 2

Let

$$\Psi(N) = \max_{P} \Phi(N,P)$$

Then

Table A.1 Values of $\Phi(N,P)$

P \ N	1	2	4	8	16	32
1	1	1	1	1	1	1
2		1	1	1	1	1
3			2	2	2	2
4			1	5	5	5
5				6	14	14
6				6	26	42
7				4	44	100
8				1	69	221
9					94	470
10					114	958
11					116	1860
12					94	3434
13					60	6036
14					28	10068
15					8	15864
16					1	23461
17						32398
18						41658
19						49700
20						54746
21						55308
22						50788
23						41944
24						30788
25						19788
26						10948
27						5096
28						1932
29						568
30						120
31						16
32						1

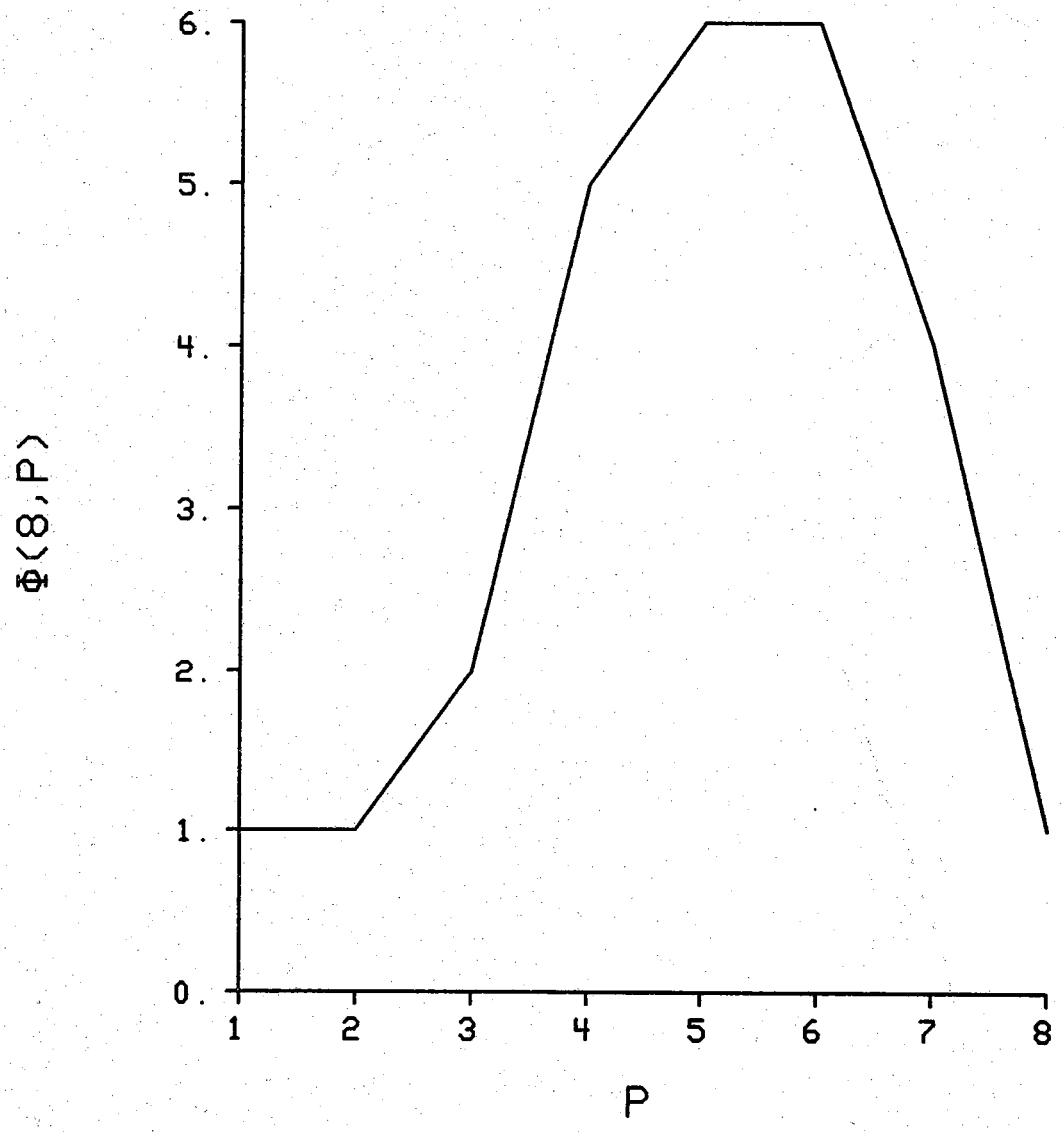


Figure A.3 $\Phi(N,P)$ for N=8

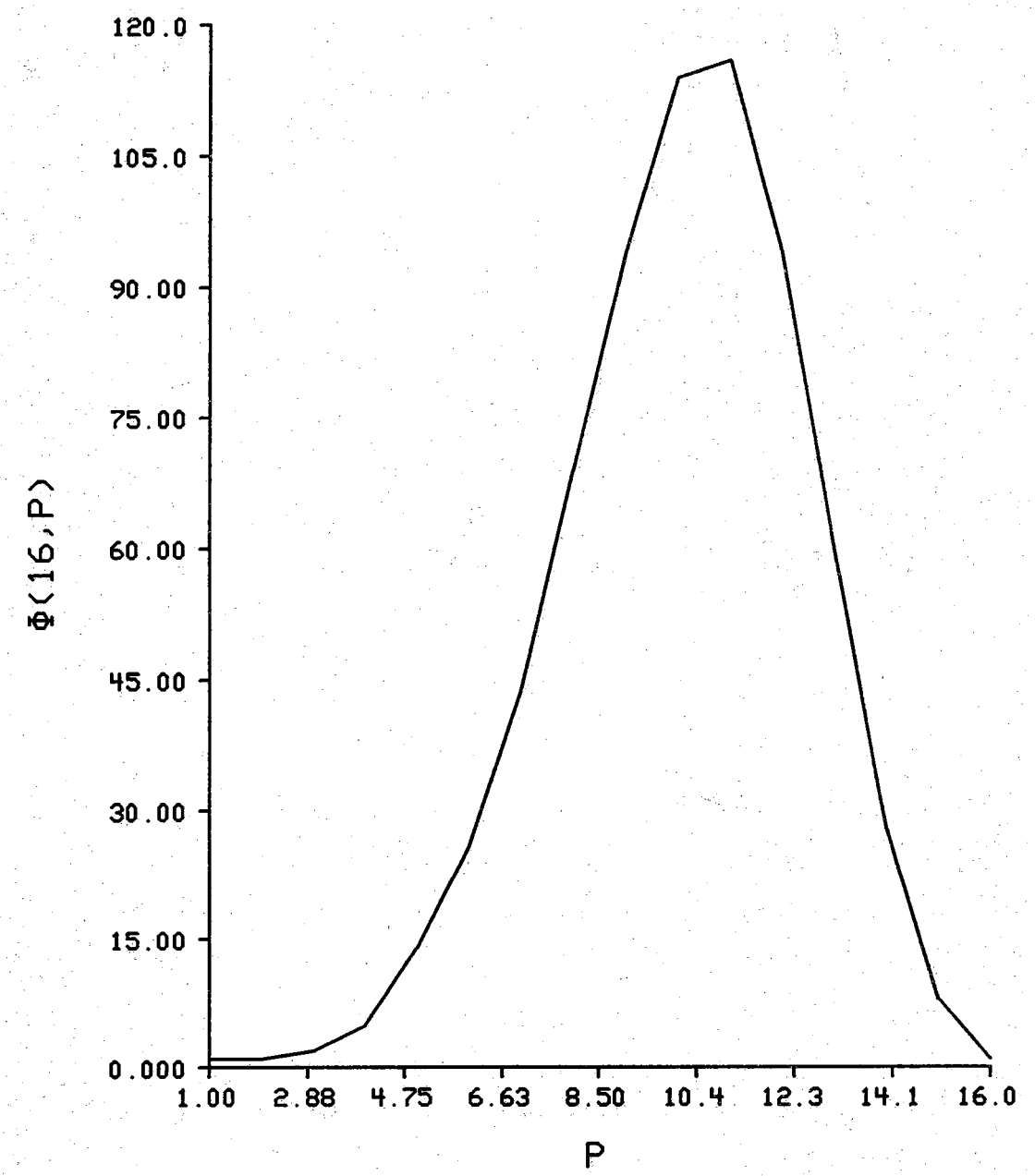


Figure A.4 Φ(N,P) for N=16

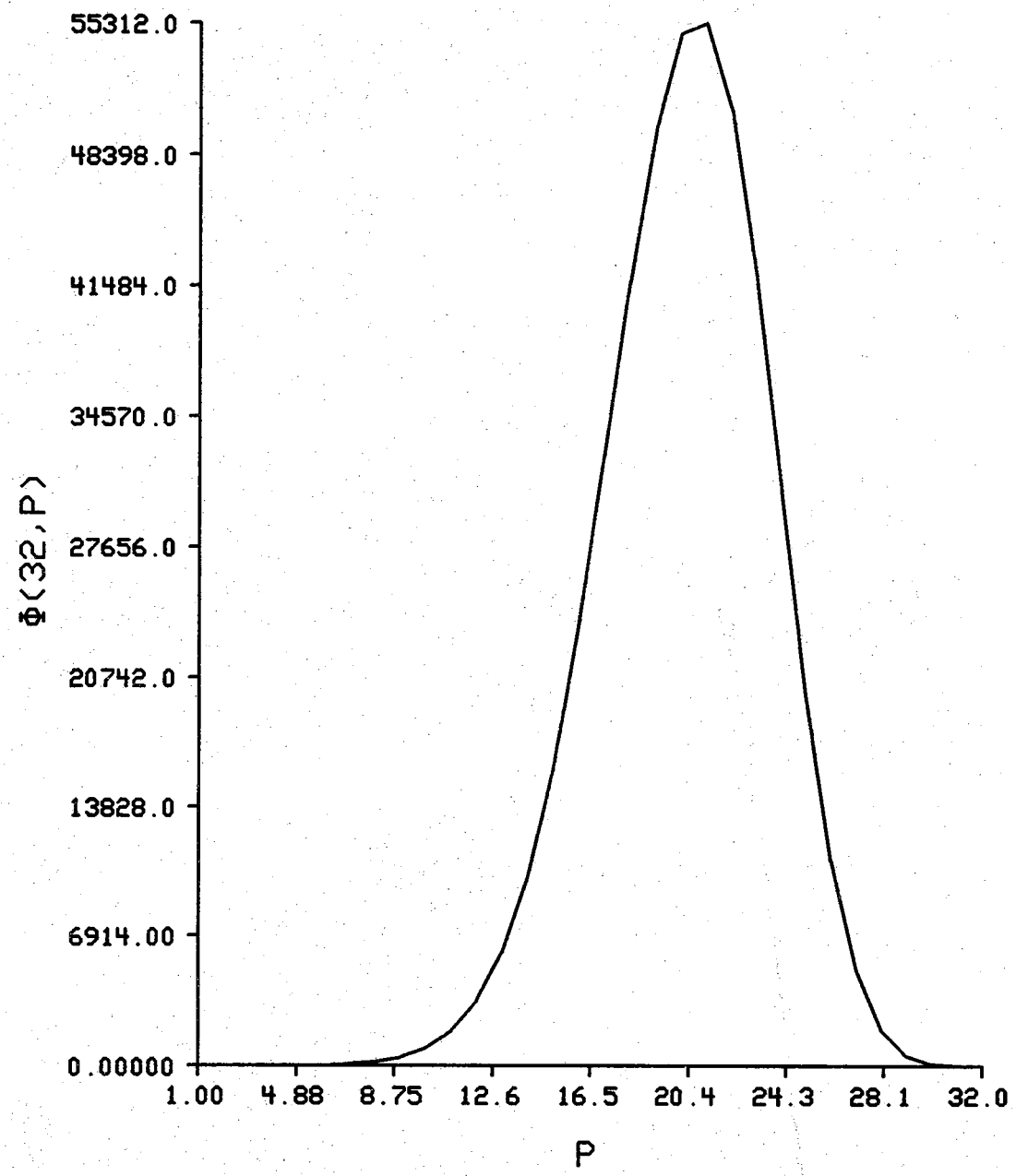


Figure A.5 Φ(N,P) for N=32

$$\Psi(N) \geqq 2^{\frac{N}{4}}$$

Proof:

Let $\Psi(N)$ occur at P=m, then $\Psi(N) = \Phi(N,m)$. Consider $\Phi(2N,2m)$, the number of partitionings possible in a machine with twice as many MCs and twice the number of partitions as $\Phi(N,P)$. We have

$$\Phi(2N,2m) = \sum_{i=1}^{2m-1} \Phi(N,i)\Phi(N,2m-i)$$

$$= \Phi(N,m)\Phi(N,m) + \sum_{\substack{i=1 \\ i\neq m}}^{2m-1} \Phi(N,i)\Phi(N,2m-1)$$

$$\geqq \Phi(N,m)^2$$

We have, then, that

$$\Psi(2N) \geqq \Phi(2N,2m) \geqq \Phi(N,m)^2$$

and

$$\Psi(N) = \Phi(N,m).$$

Therefore,

$$\Psi(2N) \geqq \Psi(N)^2.$$

Given the base condition of $\Psi(4) = 2$, the following relations can be generated:

$$\Psi(N) \geqq \Psi(\frac{N}{2})^2 \geqq \Psi(\frac{N}{4})^4 \geqq \ldots \geqq \Psi(4)^{\frac{N}{4}}$$

Therefore, we have that

$$\Psi(N) \geqq 2^{\frac{N}{4}}$$

■

Although Theorem 2 gives a lower bound on the maximum number of partitionings for a given machine size, it does not give a good indication of how $\Phi(N,P)$ behaves in general. By modifying the problem slightly, a closed form solution can be obtained.

How many binary tree are there such that

(1) All nodes of a tree have either 0 or 2 children.

(2) Given some number $P \leqq N$, there are exactly P leaf nodes in the tree.

This problem is essentially the same as before except that now there is no depth restriction on the trees. (The depth restriction of the original problem comes from the fact that we are counting subtrees of a tree with a fixed depth.)

It can be shown that the number of such trees is

$$\chi(P) = \frac{1}{P}\binom{2P-2}{P-1}$$

These numbers are referred to as *Catalan numbers* [Well71]. It can be shown that $\Phi(N,P) \leqq \chi(P)$. In order to prove this statement, another short theorem needs to be derived first. A binary tree of a given depth with the minimum possible number of nodes will be called *minimal.*

Theorem 3

A minimal binary tree of depth d has $2d-1$ nodes.

Proof (by induction):

Assume a minimal binary tree $\mathcal{G}$ of depth d has 2d−1 nodes. In order to construct a minimal binary tree $\mathcal{G}'$ of depth d+1 from $\mathcal{G}$, two children must be added to exactly one node of $\mathcal{G}$ that is already at depth d. $\mathcal{G}'$ then has

$$2d-1 + 2 = 2d+1 = 2(d+1)-1$$

nodes. The initial case is a binary tree of depth d=1. This tree has 1 node and $2 \cdot 1-1 = 1$.

■

Now the theorem about Catalan numbers can be proven.

Theorem 4

(A) $\Phi(N,P) \leqq \chi(P)$

and

(B) $\Phi(N,P) = \chi(P)$ iff $P \leqq \log N + 1$

Proof:

Part (A) is easily seen since $\chi(P)$ has no depth restriction and therefore the number of possible trees is not bounded by the depth of the tree G_M that represents the machine. There can therefore be trees counted in $\chi(P)$ that are not allowed in $\Phi(N,P)$.

For part (B), the equality holds when the maximum depth of the trees of $\chi(P)$ cannot exceed the depth of G_M. By definition, the depth of G_M is $\log N + 1$. We need to determine for what values of P can the depth of a tree exceed $\log N + 1$. That is, what is the minimum value of P for a tree with a depth of

$(\log N + 1) + 1 = \log N + 2$? By theorem 3, a minimal binary tree of depth $\log N + 2$ has $2\log N + 3$ nodes. We also know that a tree with P leaf nodes has $2P-1$ nodes in total. Equating these two equations gives

$$2P-1 = 2\log N + 3$$

$$P = \log N + 2$$

Therefore, whenever P is less than $\log N + 2$, the trees of $\chi(P)$ cannot exceed depth $\log N + 1$ and therefore $\Phi(N,P) = \chi(P)$

■

Given a function that increases exponentially or factorially, enumerating and evaluating all of its possible values could take longer than the processing time of the given task. For this reason, an enumerative generate-and-test strategy for partitioning is not feasible.

Appendix B
The CLIPS Language

This appendix will give a brief overview of the CLIPS expert system shell. The first section will point out some salient features of the CLIPS language and some of the more useful shell commands. Section 2 lists a small example expert system. The last section shows runs of the expert system with various debugging aids enabled. This appendix is not intended to be an exhaustive listing of all of CLIPS's features. The purpose, instead, is to give the reader a general idea of the general operation of CLIPS.

B.1 CLIPS Language Summary

The CLIPS language is case sensitive and forward chaining. Its inference engine uses the Rete algorithm [Forg82] to match the facts and rules. CLIPS is composed of only two constructs: *facts* and *rules*.

Facts are asserted into the internal fact base through the *deffacts* directive when the system is initialized. It has the following form:

```
(deffacts <name> <comment>
        (fact 1)
        (fact 2)
        ⋮
        (fact n))
```

The <name> and <comment> fields are optional. The zero or more facts can be any sequence of words, characters, numbers, or quoted strings (with the exception of the variable characters *?* and *$?*). There can be any number of deffacts statements. All facts in the following example are legal and distinct.

```
(this is a fact)
(This is a fact)
(MORE FACTS)
("MORE FACTS")
(_some_data 9 37.3 -1.54e-13)
(K9 "is a" dog)
```

The rules are translated into an internal form when they are read in and are fired by the inference engine at runtime. Rules have the following form:

```
(defrule <name> <comment>
      (LHS 1)
      ⋮
      (LHS m)
   =>
      (RHS 1)
      ⋮
      (RHS n))
```

As with deffacts constructs, the <comment> field is optional. The <name> field must be present. LHS is an abbreviation for Left Hand Side and includes all items between the 'defrule' line and the '=>'. These items are the conditions that have to be satisfied before the rule will fire. RHS is an abbreviation for Right Hand Side and includes all items between the '=>' and the closing ')'. These items are the actions that will be taken once the rule fires.

LHS items can be one of three types of statements. The first type is a directive to the inference engine. These directives take the form of a 'declare' statement. There is only one directive currently implemented. It informs the inference engine of the priority of the rule and has the form '(declare (salience ###))' where '###' is the priority (or *salience*) of the rule and can range from -10000 to 10000.

The second type of LHS statement is a fact to be matched. It has the same form as the facts mentioned previously with one exception. That exception is that facts to be matched in the LHS of a rule can have wildcards and variables in place of the regular words, numbers, etc. The two wildcard indicators are '?' and '$?'. A '?' will match any single field within a fact. For example, (junk ?) will match both (junk hello) and (junk 10.6) but not (junk) or (junk yard dog). A '$?' will match any sequence of zero or more fields within a fact. For example, (facts $?) will match (facts abc), (facts 15 more "hello there" 88) and (facts). A variable name can be appended to '?' or '$?', and the name will be bound to the fields that matched the wildcards. For example, the pattern

(? stuff ?var1 $?var2 mark $? again ?more)

will match the fact

(anything stuff -555 a few "fields here" mark a b c 1 2 34.56 again last)

and the following bindings will result:

var1 ← -555, var2 ← 'a few "fields here"', more ← last

These variables can then be used in other LHS fact patterns and in RHS actions.

The third type of LHS statement is functional operations. Any built-in or user-defined functions can be used. CLIPS contains a built-in extended math function library comprised of functions such as *min, max, sqrt, exp, sin, acosh,* etc. These functions can either be used within a fact pattern (which will not be discussed here) or as part of a *test* statement. A 'test' statement has the syntax: (test <any_function>). For example,

(test (|| (> ?a 10) (neq ?first ?second)))

will return TRUE if either 'a' is greater than 10 or 'first' is not equal to 'second' (functions use prefix notation in CLIPS).

The RHS of a rule contains a list of all actions to be performed when the rule fires. These actions can be any built-in or user-written function. CLIPS has a large number of such functions, but only a few will be mentioned here. The *assert* function is used to add new facts to the fact base. The basic syntax is (assert <fact>) where fact is as before. Variables can be used as fields in the fact. For example, if 'data' is bound to 100.567, then (assert (numbers ?data)) will result in the fact (numbers 100.567) being added to the fact base.

The complement to 'assert' is *retract*, which removes facts from the fact base. The usage of 'retract' is (retract fact1 fact2 ... factn). The only facts that can be removed by the 'retract' statement are those that were matched in the LHS of the rule. In order to remove on of these facts, a special syntax is used in the LHS to assign the fact number to a variable.
For example, consider the following rule:

```
(defrule example_1
   ?fnum <- (some data 1.0)
=>
   (retract ?fnum))
```

When this rule fires, the fact (some data 1.0) will be removed from the fact base. The special symbol '<-' is used to access the fact number in the LHS of the rule.

Any rule can access user-written functions by directly calling the function. For example, suppose the user has written and linked in a function called 'calc' which calculates values of the function

$$x^2 + 3xy + y^2 - 12.$$

The function return value could be bound to the variable 'result' by the *bind* function as follows: (bind ?result (calc 3 4)) where the first parameter will be used as x and the second will be used for y. The

variable 'result' is bound to 49.

Comments in CLIPS are designated by a ';'. Anything from a ';' to the end of a line is ignored by the rule interpreter. These comments can occur anywhere in a 'defrule' of 'deffacts' construct except within a quoted string.

The CLIPS shell contains many commands for execution control, debugging, and system prototyping. Table B.1 lists some of the more commonly used commands and a brief description of their function.

B.2 Example Expert System

The following expert system was written at NASA as part of the test package that accompanies CLIPS. It presents a relatively short ES (only 10 rules) whose rules have a range of complexities. The last section will show various runs of this ES.

Table B.1 CLIPS Shell Commands

COMMAND	FUNCTION
dribble-on, dribble-off	Starts and stops the storage in a file of all information sent to the screen.
exit	Quits the CLIPS environment.
facts	Lists the facts in the fact base.
help	Gets information from the on-line help system.
load	Loads the facts and rules in a specified file into the environment.
reset	Resets CLIPS. Removes all activations and facts and then asserts all facts from deffacts statements.
rules	Lists the rules in the environment.
run	Starts execution of the rules.
watch, unwatch	Starts and stops the tracing of rule firings and activations and fact assertions and retractions during a run.
<function>	Any built-in or user-defined function can be called from the shell.

```
;;;=========================================================
;;;   Farmer's Dilemma Problem
;;;
;;;     Another classic AI problem (cannibals and the
;;;     missionary) in agricultural terms. The point is
;;;     to get the farmer, the fox the cabbage and the
;;;     goat across a stream.
;;;     But the boat only holds 2 items. If left
;;;     alone with the goat, the fox will eat it. If
;;;     left alone with the cabbage, the goat will eat
;;;     it.
;;;
;;;     To execute, merely load, reset and run.
;;;=========================================================

;;;*****************************
;;;* Farmer's Dilemma Problem *
;;;*****************************

;;; The status facts hold the state information
;;; of the search tree.
;;; (status <search-depth>
;;;         <id>
;;;         <parent>
;;;         <farmer-location>
;;;         <fox-location>
;;;         <goat-location>
;;;         <cabbage-location>)

;;; The moves facts hold the information of all the moves
;;; made to reach a given state.
;;; (status <id>
;;;         <move-1>
;;;            .
;;;            .
;;;            .
;;;         <move-n>)

;;;******************
;;;* Initial State *
;;;******************

(deffacts initial-positions
```

```
   (status 1 initial-setup no-parent shore-1
           shore-1 shore-1 shore-1 no-move))

(deffacts opposites
   (opposite-of shore-1 shore-2)
   (opposite-of shore-2 shore-1))

;;;*************************
;;;* Generate Paths Rules *
;;;*************************

(defrule move-alone ""
   (status ?num ?name ? ?fs ?xs ?gs ?cs ?)
   (opposite-of ?fs ?ns)
   =>
   (bind ?nn (gensym))
   (assert (status =(+ 1 ?num)
            ?nn ?name ?ns ?xs ?gs ?cs alone)))

(defrule move-with-fox ""
   (status ?num ?name ? ?fs ?fs ?gs ?cs ?)
   (opposite-of ?fs ?ns)
   =>
   (bind ?nn (gensym))
   (assert (status =(+ 1 ?num)
            ?nn ?name ?ns ?ns ?gs ?cs fox)))

(defrule move-with-goat ""
   (status ?num ?name ? ?fs ?xs ?fs ?cs ?)
   (opposite-of ?fs ?ns)
   =>
   (bind ?nn (gensym))
   (assert (status =(+ 1 ?num)
            ?nn ?name ?ns ?xs ?ns ?cs goat)))

(defrule move-with-cabbage ""
   (status ?num ?name ? ?fs ?xs ?gs ?fs ?)
   (opposite-of ?fs ?ns)
   =>
   (bind ?nn (gensym))
   (assert (status =(+ 1 ?num)
            ?nn ?name ?ns ?xs ?gs ?ns cabbage)))
```

```
;;;*******************************
;;;* Constraint Violation Rules *
;;;*******************************

(defrule fox-eats-goat ""
  (declare (salience 10000))
  ?rm <- (status ? ?name ? ?s1 ?s2&~?s1 ?s2 ? ?)
  =>
  (retract ?rm))

(defrule goat-eats-cabbage ""
  (declare (salience 10000))
  ?rm <- (status ? ?name ? ?s1 ? ?s2&~?s1 ?s2 ?)
  =>
  (retract ?rm))

(defrule circular-path ""
  (declare (salience 10000))
  (status ?nm ? ? ?fs ?xs ?gs ?cs ?)
  ?rm <- (status ?nm1&:(< ?nm ?nm1) ?name ?
          ?fs ?xs ?gs ?cs ?)
  =>
  (retract ?rm))

;;;*********************************
;;;* Find and Print Solution Rule *
;;;*********************************

(defrule recognize-solution ""
  (declare (salience 5000))
  ?rm <- (status ?num ?name ?parent shore-2
          shore-2 shore-2 shore-2 ?move)
  =>
  (retract ?rm)
  (assert (moves ?parent ?move)))

(defrule further-solution ""
  (declare (salience 5000))
  ?mv <- (moves ?name $?rest)
  (status ? ?name ?parent ? ? ? ? ?move)
  =>
  (retract ?mv)
  (assert (moves ?parent ?move $?rest)))
```

```
(defrule print-solution ""
  (declare (salience 5000))
  ?mv <- (moves no-parent no-move $?m)
  =>
  (retract ?mv)
  (printout t t  "Solution found: " t t)
  (bind ?length (length $?m))
  (bind ?i 1)
  (bind ?shore shore-2)
  (while (<= ?i ?length)
     (bind ?thing (nth ?i $?m))
     (if (eq ?thing alone)
        then
         (printout t "Farmer moves alone to " ?shore "." t)
        else
         (printout t "Farmer moves with " ?thing "
                     to " ?shore "." t))
     (if (eq ?shore shore-1)
        then (bind ?shore shore-2)
        else (bind ?shore shore-1))
     (bind ?i (+ 1 ?i))))
```

B.3 Example Expert System Runs

This section presents two runs of the previous expert system. The first run has no extra tracing enabled. The output produced is the default for the shell. The second run has rule firing and fact tracing enabled. Note that the rules are not fired in sequential order. Even for those rules with the same salience value, execution order does not follow the ordering of the rules in the input file.

The CLIPS shell prompt is 'CLIPS> '. All shell commands are surrounded by ()'s. Any lines that do not start with the prompt are output from the ES or the shell. After the 'load' command, there is a line of dollar signs and asterisks. For each fact loaded, a '$' is printed. For each rule loaded, a '*' is printed.

```
         CLIPS (V4.20 4/29/88)
CLIPS> (load "dilemma.clp")
$$**********
CLIPS> (reset)
CLIPS> (facts)
f-0     (initial-fact)
f-1     (status 1 initial-setup no-parent shore-1 shore-1
         shore-1 shore-1 no-move)
f-2     (opposite-of shore-1 shore-2)
f-3     (opposite-of shore-2 shore-1)
CLIPS> (rules)
move-alone
move-with-fox
move-with-goat
move-with-cabbage
fox-eats-goat
goat-eats-cabbage
circular-path
recognize-solution
further-solution
print-solution
CLIPS> (run)

Solution found:

Farmer moves with goat to shore-2.
Farmer moves alone to shore-1.
Farmer moves with cabbage to shore-2.
Farmer moves with goat to shore-1.
Farmer moves with fox to shore-2.
Farmer moves alone to shore-1.
Farmer moves with goat to shore-2.

Solution found:

Farmer moves with goat to shore-2.
Farmer moves alone to shore-1.
Farmer moves with fox to shore-2.
Farmer moves with goat to shore-1.
Farmer moves with cabbage to shore-2.
Farmer moves alone to shore-1.
Farmer moves with goat to shore-2.
80 rules fired
```

```
CLIPS> (facts)
f-0     (initial-fact)
f-1     (status 1 initial-setup no-parent shore-1 shore-1
         shore-1 shore-1 no-move)
f-2     (opposite-of shore-1 shore-2)
f-3     (opposite-of shore-2 shore-1)
f-6     (status 2 gen3 initial-setup shore-2 shore-1
         shore-2 shore-1 goat)
f-7     (status 3 gen4 gen3 shore-1 shore-1 shore-2
         shore-1 alone)
f-9     (status 4 gen6 gen4 shore-2 shore-1 shore-2
         shore-2 cabbage)
f-12    (status 5 gen9 gen6 shore-1 shore-1 shore-1
         shore-2 goat)
f-15    (status 6 gen12 gen9 shore-2 shore-2 shore-1
         shore-2 fox)
f-16    (status 7 gen13 gen12 shore-1 shore-2 shore-1
         shore-2 alone)
f-35    (status 4 gen24 gen4 shore-2 shore-2 shore-2
         shore-1 fox)
f-37    (status 5 gen26 gen24 shore-1 shore-2 shore-1
         shore-1 goat)
f-39    (status 6 gen28 gen26 shore-2 shore-2 shore-1
         shore-2 cabbage)
f-40    (status 7 gen29 gen28 shore-1 shore-2 shore-1
         shore-2 alone)
```

```
CLIPS> (reset)
CLIPS> (watch rules)
CLIPS> (watch facts)
CLIPS> (facts)
f-0     (initial-fact)
f-1     (status 1 initial-setup no-parent shore-1 shore-1
         shore-1 shore-1 no-move)
f-2     (opposite-of shore-1 shore-2)
f-3     (opposite-of shore-2 shore-1)
CLIPS> (run)
FIRE    1 move-alone: f-1,f-2
==> f-4     (status 2 gen38 initial-setup shore-2 shore-1
             shore-1 shore-1 alone)
FIRE    2 goat-eats-cabbage: f-4
<== f-4     (status 2 gen38 initial-setup shore-2 shore-1
             shore-1 shore-1 alone)
FIRE    3 move-with-fox: f-1,f-2
==> f-5     (status 2 gen39 initial-setup shore-2 shore-2
             shore-1 shore-1 fox)
FIRE    4 goat-eats-cabbage: f-5
<== f-5     (status 2 gen39 initial-setup shore-2 shore-2
             shore-1 shore-1 fox)
FIRE    5 move-with-goat: f-1,f-2
==> f-6     (status 2 gen40 initial-setup shore-2 shore-1
             shore-2 shore-1 goat)
FIRE    6 move-alone: f-6,f-3
==> f-7     (status 3 gen41 gen40 shore-1 shore-1 shore-2
             shore-1 alone)
FIRE    7 move-alone: f-7,f-2
==> f-8     (status 4 gen42 gen41 shore-2 shore-1 shore-2
             shore-1 alone)
FIRE    8 circular-path: f-6,f-8
<== f-8     (status 4 gen42 gen41 shore-2 shore-1 shore-2
             shore-1 alone)
FIRE    9 move-with-cabbage: f-7,f-2
==> f-9     (status 4 gen43 gen41 shore-2 shore-1 shore-2
             shore-2 cabbage)
FIRE   10 move-alone: f-9,f-3
==> f-10    (status 5 gen44 gen43 shore-1 shore-1 shore-2
             shore-2 alone)
FIRE   11 goat-eats-cabbage: f-10
<== f-10    (status 5 gen44 gen43 shore-1 shore-1 shore-2
             shore-2 alone)
```

```
FIRE    12 move-with-cabbage: f-9,f-3
==> f-11    (status 5 gen45 gen43 shore-1 shore-1 shore-2
             shore-1 cabbage)
FIRE    13 circular-path: f-7,f-11
<== f-11    (status 5 gen45 gen43 shore-1 shore-1 shore-2
             shore-1 cabbage)
FIRE    14 move-with-goat: f-9,f-3
==> f-12    (status 5 gen46 gen43 shore-1 shore-1 shore-1
              shore-2 goat)
FIRE    15 move-alone: f-12,f-2
==> f-13    (status 6 gen47 gen46 shore-2 shore-1 shore-1
             shore-2 alone)
FIRE    16 fox-eats-goat: f-13
<== f-13    (status 6 gen47 gen46 shore-2 shore-1 shore-1
             shore-2 alone)
FIRE    17 move-with-goat: f-12,f-2
==> f-14    (status 6 gen48 gen46 shore-2 shore-1 shore-2
             shore-2 goat)
FIRE    18 circular-path: f-9,f-14
<== f-14    (status 6 gen48 gen46 shore-2 shore-1 shore-2
             shore-2 goat)
FIRE    19 move-with-fox: f-12,f-2
==> f-15    (status 6 gen49 gen46 shore-2 shore-2 shore-1
             shore-2 fox)
FIRE    20 move-alone: f-15,f-3
==> f-16    (status 7 gen50 gen49 shore-1 shore-2 shore-1
             shore-2 alone)
FIRE    21 move-alone: f-16,f-2
==> f-17    (status 8 gen51 gen50 shore-2 shore-2 shore-1
             shore-2 alone)
FIRE    22 circular-path: f-15,f-17
<== f-17    (status 8 gen51 gen50 shore-2 shore-2 shore-1
             shore-2 alone)
FIRE    23 move-with-goat: f-16,f-2
==> f-18    (status 8 gen52 gen50 shore-2 shore-2 shore-2
             shore-2 goat)
FIRE    24 recognize-solution: f-18
<== f-18    (status 8 gen52 gen50 shore-2 shore-2 shore-2
             shore-2 goat)
==> f-19    (moves gen50 goat)
FIRE    25 further-solution: f-19,f-16
<== f-19    (moves gen50 goat)
==> f-20    (moves gen49 alone goat)
```

```
FIRE    26 further-solution: f-20,f-15
<== f-20     (moves gen49 alone goat)
==> f-21     (moves gen46 fox alone goat)
FIRE    27 further-solution: f-21,f-12
<== f-21     (moves gen46 fox alone goat)
==> f-22     (moves gen43 goat fox alone goat)
FIRE    28 further-solution: f-22,f-9
<== f-22     (moves gen43 goat fox alone goat)
==> f-23     (moves gen41 cabbage goat fox alone goat)
FIRE    29 further-solution: f-23,f-7
<== f-23     (moves gen41 cabbage goat fox alone goat)
==> f-24     (moves gen40 alone cabbage goat fox
              alone goat)
FIRE    30 further-solution: f-24,f-6
<== f-24     (moves gen40 alone cabbage goat fox
              alone goat)
==> f-25     (moves initial-setup goat alone cabbage
              goat fox alone goat)
FIRE    31 further-solution: f-25,f-1
<== f-25     (moves initial-setup goat alone cabbage
              goat fox alone goat)
==> f-26     (moves no-parent no-move goat alone cabbage
              goat fox alone goat)
FIRE    32 print-solution: f-26
<== f-26     (moves no-parent no-move goat alone cabbage
              goat fox alone goat)

Solution found:

Farmer moves with goat to shore-2.
Farmer moves alone to shore-1.
Farmer moves with cabbage to shore-2.
Farmer moves with goat to shore-1.
Farmer moves with fox to shore-2.
Farmer moves alone to shore-1.
Farmer moves with goat to shore-2.
FIRE    33 move-with-cabbage: f-15,f-3
==> f-27     (status 7 gen53 gen49 shore-1 shore-2 shore-1
              shore-1 cabbage)
FIRE    34 move-alone: f-27,f-2
==> f-28     (status 8 gen54 gen53 shore-2 shore-2 shore-1
              shore-1 alone)
FIRE    35 goat-eats-cabbage: f-28
```

```
<== f-28     (status 8 gen54 gen53 shore-2 shore-2 shore-1
              shore-1 alone)
FIRE    36 move-with-cabbage: f-27,f-2
==> f-29     (status 8 gen55 gen53 shore-2 shore-2 shore-1
              shore-2 cabbage)
FIRE    37 circular-path: f-15,f-29
<== f-29     (status 8 gen55 gen53 shore-2 shore-2 shore-1
              shore-2 cabbage)
FIRE    38 move-with-goat: f-27,f-2
==> f-30     (status 8 gen56 gen53 shore-2 shore-2 shore-2
              shore-1 goat)
FIRE    39 move-alone: f-30,f-3
==> f-31     (status 9 gen57 gen56 shore-1 shore-2 shore-2
              shore-1 alone)
FIRE    40 fox-eats-goat: f-31
<== f-31     (status 9 gen57 gen56 shore-1 shore-2 shore-2
              shore-1 alone)
FIRE    41 move-with-goat: f-30,f-3
==> f-32     (status 9 gen58 gen56 shore-1 shore-2 shore-1
              shore-1 goat)
FIRE    42 circular-path: f-27,f-32
<== f-32     (status 9 gen58 gen56 shore-1 shore-2 shore-1
              shore-1 goat)
FIRE    43 move-with-fox: f-30,f-3
==> f-33     (status 9 gen59 gen56 shore-1 shore-1 shore-2
              shore-1 fox)
FIRE    44 circular-path: f-7,f-33
<== f-33     (status 9 gen59 gen56 shore-1 shore-1 shore-2
              shore-1 fox)
FIRE    45 move-with-fox: f-15,f-3
==> f-34     (status 7 gen60 gen49 shore-1 shore-1 shore-1
              shore-2 fox)
FIRE    46 circular-path: f-12,f-34
<== f-34     (status 7 gen60 gen49 shore-1 shore-1 shore-1
              shore-2 fox)
FIRE    47 move-with-fox: f-7,f-2
==> f-35     (status 4 gen61 gen41 shore-2 shore-2 shore-2
              shore-1 fox)
FIRE    48 circular-path: f-35,f-30
<== f-30     (status 8 gen56 gen53 shore-2 shore-2 shore-2
              shore-1 goat)
FIRE    49 move-alone: f-35,f-3
==> f-36     (status 5 gen62 gen61 shore-1 shore-2 shore-2
```

```
            shore-1 alone)
FIRE    50 fox-eats-goat: f-36
<== f-36    (status 5 gen62 gen61 shore-1 shore-2 shore-2
             shore-1 alone)
FIRE    51 move-with-goat: f-35,f-3
==> f-37    (status 5 gen63 gen61 shore-1 shore-2 shore-1
             shore-1 goat)
FIRE    52 circular-path: f-37,f-27
<== f-27    (status 7 gen53 gen49 shore-1 shore-2 shore-1
             shore-1 cabbage)
FIRE    53 move-alone: f-37,f-2
==> f-38    (status 6 gen64 gen63 shore-2 shore-2 shore-1
             shore-1 alone)
FIRE    54 goat-eats-cabbage: f-38
<== f-38    (status 6 gen64 gen63 shore-2 shore-2 shore-1
             shore-1 alone)
FIRE    55 move-with-cabbage: f-37,f-2
==> f-39    (status 6 gen65 gen63 shore-2 shore-2 shore-1
             shore-2 cabbage)
FIRE    56 move-alone: f-39,f-3
==> f-40    (status 7 gen66 gen65 shore-1 shore-2 shore-1
             shore-2 alone)
FIRE    57 move-alone: f-40,f-2
==> f-41    (status 8 gen67 gen66 shore-2 shore-2 shore-1
             shore-2 alone)
FIRE    58 circular-path: f-15,f-41
<== f-41    (status 8 gen67 gen66 shore-2 shore-2 shore-1
             shore-2 alone)
FIRE    59 move-with-goat: f-40,f-2
==> f-42    (status 8 gen68 gen66 shore-2 shore-2 shore-2
             shore-2 goat)
FIRE    60 recognize-solution: f-42
<== f-42    (status 8 gen68 gen66 shore-2 shore-2 shore-2
             shore-2 goat)
==> f-43    (moves gen66 goat)
FIRE    61 further-solution: f-43,f-40
<== f-43    (moves gen66 goat)
==> f-44    (moves gen65 alone goat)
FIRE    62 further-solution: f-44,f-39
<== f-44    (moves gen65 alone goat)
==> f-45    (moves gen63 cabbage alone goat)
FIRE    63 further-solution: f-45,f-37
<== f-45    (moves gen63 cabbage alone goat)
```

```
==> f-46    (moves gen61 goat cabbage alone goat)
FIRE   64 further-solution: f-46,f-35
<== f-46    (moves gen61 goat cabbage alone goat)
==> f-47    (moves gen41 fox goat cabbage alone goat)
FIRE   65 further-solution: f-47,f-7
<== f-47    (moves gen41 fox goat cabbage alone goat)
==> f-48    (moves gen40 alone fox goat cabbage alone goat)
FIRE   66 further-solution: f-48,f-6
<== f-48    (moves gen40 alone fox goat cabbage alone goat)
==> f-49    (moves initial-setup goat alone fox goat cabbage
              alone goat)
FIRE   67 further-solution: f-49,f-1
<== f-49    (moves initial-setup goat alone fox goat cabbage
             alone goat)
==> f-50    (moves no-parent no-move goat alone fox goat
             cabbage alone goat)
FIRE   68 print-solution: f-50
<== f-50    (moves no-parent no-move goat alone fox goat
             cabbage alone goat)

Solution found:

Farmer moves with goat to shore-2.
Farmer moves alone to shore-1.
Farmer moves with fox to shore-2.
Farmer moves with goat to shore-1.
Farmer moves with cabbage to shore-2.
Farmer moves alone to shore-1.
Farmer moves with goat to shore-2.
FIRE   69 move-with-cabbage: f-39,f-3
==> f-51    (status 7 gen69 gen65 shore-1 shore-2 shore-1
             shore-1 cabbage)
FIRE   70 circular-path: f-37,f-51
<== f-51    (status 7 gen69 gen65 shore-1 shore-2 shore-1
             shore-1 cabbage)
FIRE   71 move-with-fox: f-39,f-3
==> f-52    (status 7 gen70 gen65 shore-1 shore-1 shore-1
             shore-2 fox)
FIRE   72 circular-path: f-12,f-52
<== f-52    (status 7 gen70 gen65 shore-1 shore-1 shore-1
             shore-2 fox)
FIRE   73 move-with-goat: f-37,f-2
==> f-53    (status 6 gen71 gen63 shore-2 shore-2 shore-2
```

```
             shore-1 goat)
FIRE    74 circular-path: f-35,f-53
<== f-53     (status 6 gen71 gen63 shore-2 shore-2 shore-2
             shore-1 goat)
FIRE    75 move-with-fox: f-35,f-3
==> f-54     (status 5 gen72 gen61 shore-1 shore-1 shore-2
             shore-1 fox)
FIRE    76 circular-path: f-7,f-54
<== f-54     (status 5 gen72 gen61 shore-1 shore-1 shore-2
             shore-1 fox)
FIRE    77 move-with-goat: f-6,f-3
==> f-55     (status 3 gen73 gen40 shore-1 shore-1 shore-1
             shore-1 goat)
FIRE    78 circular-path: f-1,f-55
<== f-55     (status 3 gen73 gen40 shore-1 shore-1 shore-1
             shore-1 goat)
FIRE    79 move-with-cabbage: f-1,f-2
==> f-56     (status 2 gen74 initial-setup shore-2 shore-1
             shore-1 shore-2 cabbage)
FIRE    80 fox-eats-goat: f-56
<== f-56     (status 2 gen74 initial-setup shore-2 shore-1
          shore-1 shore-2 cabbage)
80 rules fired
CLIPS> (facts)
f-0      (initial-fact)
f-1      (status 1 initial-setup no-parent shore-1
          shore-1 shore-1 shore-1 no-move)
f-2      (opposite-of shore-1 shore-2)
f-3      (opposite-of shore-2 shore-1)
f-6      (status 2 gen40 initial-setup shore-2 shore-1
          shore-2 shore-1 goat)
f-7      (status 3 gen41 gen40 shore-1 shore-1 shore-2
          shore-1 alone)
f-9      (status 4 gen43 gen41 shore-2 shore-1 shore-2
          shore-2 cabbage)
f-12     (status 5 gen46 gen43 shore-1 shore-1 shore-1
          shore-2 goat)
f-15     (status 6 gen49 gen46 shore-2 shore-2 shore-1
          shore-2 fox)
f-16     (status 7 gen50 gen49 shore-1 shore-2 shore-1
          shore-2 alone)
f-35     (status 4 gen61 gen41 shore-2 shore-2 shore-2
          shore-1 fox)
```

```
f-37    (status 5 gen63 gen61 shore-1 shore-2 shore-1
         shore-1 goat)
f-39    (status 6 gen65 gen63 shore-2 shore-2 shore-1
         shore-2 cabbage)
f-40    (status 7 gen66 gen65 shore-1 shore-2 shore-1
         shore-2 alone)
CLIPS> (exit)
```

Appendix C
DISC System Setup

This appendix will present the layout of the directories in which the DISC system resides, give instructions for the execution of the simulation system, and provide listings of the major code sections and input files. An example run is shown from start to finish using example task II from chapter 7. The entire DISC system is comprised of about 67 rules and fact blocks (containing about 1200 lines of CLIPS code) and also about 2500 lines of support code written in C.

C.1 Directory Layout

The code for DISC is split among four directories. These directories correspond to functional divisions in the DISC system. The main directory contains the executable code (either the CLIPS shell or the compiled version), the startup code for DISC, and the task parameters file. The data dependency graph specifications for the example tasks are also in the main directory, but they do not have to be there.

The main directory has three subdirectories. The DB subdirectory contains all the database information on the algorithm implementations. The RULES subdirectory contains the CLIPS expert system code for DISC. The SRC subdirectory contains all support C code that has been compiled into CLIPS.

C.2 DISC File Examples and Instructions

This section will discuss the contents and operation of the various files and programs. Source code listings are provided at the end of this appendix and are in alphabetic order by file name. An example run of the DISC simulation system is shown at the end of this section. All instructions given here are applicable to the interpreted system.

The simulator must be run from the main directory. To initiate a run, simply type 'clips' at the operating system prompt. Once CLIPS has been started, the expert system is loaded and the simulator started by typing '(load "disc")' at the CLIPS prompt. All necessary information is then prompted for. To end a simulation session and exit CLIPS, type '(exit)' at the CLIPS prompt. The simulation output is saved in a file called 'run' in the main directory; it is overwritten at each session.

The parameters for the task such as number of pixels, number of histogram bins, etc. are read from a file named 'task.parameters' in the main directory. Each parameter conforms to CLIPS fact syntax. The format is '(Parameter <param> <value>)' where <param> is the parameter name and <value> is the parameter value. This file also contains two values used by DISC. The first is the loop weight value (giving the priority value given to loops in the DDG) and the second is the weight of data loading (indicating how much faster it is in general to reallocate or reformat the data in a partition than to reload it from mass storage).

The data dependency graph is not intended to be a human interface in the image understanding environment and, in the interests of speed, has a strict format. Each node of the DDG is identified by a unique integer to eliminate confusion if the same algorithm is used more than once in the task. Any time a node loops back to a node previously processed, the destination node must have a smaller node number. Nodes are listed sequentially in the file and the node format is:

<node number>
<node type> <node priority>
<algorithm name>
<number of parameters> <parameter list>
<children list>

The <node type> is either 'A' for a simple algorithm node or 'C' for a condition node. The <node priority> is an integer specifying the priority of the node with 0 meaning "no extra priority". The <algorithm name> must be exactly as is listed in the database. The <parameter list> consists of <number of parameters> pairs of <parameter name> and <parameter type>. A <parameter name> can be a variable name or a constant. Constants are indicated by prepending the name with an underscore. The <parameter type> can be 'I' for an input, 'M' for a modified input, 'O' for an output, or 'C' for the condition value of a condition node. If the node is type 'A', the <children list> consists of the number of children of the node and a list of those child nodes. For type 'C' nodes, the <children list> consists of the condition itself, the number of TRUE children, the TRUE child list, the number of FALSE children, and the FALSE child list. There must be no carriage return character after the final node listing.

The DB directory contains all the algorithm database files and an extra template file (db.template). Each file is named with 'db.' followed by the algorithm name. Each fact in a database file follows the CLIPS fact syntax. The file for an algorithm contains the information for every implementation of the algorithm. The ordering of the facts within the file does not matter.

An example database is included in the listings at the end of the appendix. The format of the information for a single implementation is as follows.

(DB alg_name "var_meaning" variable meaning)

(DB alg_name imp_name)
(DB imp_name "mode" mode)
(DB imp_name "num_pe" #_PEs)
(DB imp_name "time 32" time)
(DB imp_name "time 64" time)
(DB imp_name "time 128" time)
(DB imp_name "time 256" time)
(DB imp_name "time 512" time)
(DB imp_name "time 1024" time)
(DB imp_name "in_format" input_format ...)
(DB imp_name "out_format" output_format ...)
(DB imp_name "in_alloc" input_allocation ...)
(DB imp_name "out_alloc" output_allocation ...)

The "var_meaning" fact relates variable names from the execution time equations to the parameters from the 'task.parameters' file. There is one such fact for each variable. The rest of the facts are repeated once for each implementation listed in the file. The parameters "alg_name" and "imp_name" state the name of the algorithm and implementation. The mode is either "s" for SIMD or "m" for MIMD. The number of PEs required, "#_PEs", is the operator "<=" or ">=" followed by either a number or variable name. This fact places a restriction on the possible sizes for the candidate partition. The "time" parameter gives a symbolic equation for the expected execution time. Variables, numbers, parentheses, the arithmetic operators (+, -, *, /), and "l a b" for "log base a of b" can be used in the equations. The equations are in prefix (LISP-like) notation. If the execution time is unknown, a "1" is used for the equation. The "input_format" and "output_format" list the format of each input, modified input, and output. Modified inputs are considered both inputs and outputs. The "input_allocation" and "output_allocation" list the allocation of data to PEs for all inputs, modified inputs, and outputs. For both format and allocation listings, the specification can be any ASCII string. Appropriate strings might be, for example, "integer" or "column." The actual string that is used

does not matter since DISC does a simple string comparison when formats or allocations need to be compared. The only requirement is that the same unique string must always be used to indicate a particular format or allocation.

The RULES directory contains all the source listings of the DISC expert system. The format of the file names is "disc.*section*.rules" where *section* is "cleanup," "init," "load," "main1," "main2," "main3," or "main4." These sections were discussed in Chapter 6 and will not be covered here.

The SRC directory contains the C source code for the embedded functions used by DISC. The following list names the source files and gives a brief description of their functions.

assert_RDDG.c	Assert the reduced data dependency graph into the CLIPS fact base.
finished.c	Check to see if any algorithm has finished. (Used mostly for simulation purposes.)
initialize.c	Set up the processing of the data dependency graph.
letter.c	Fast way to return the first letter of a character string. (Used to quickly identify constants by looking for an initial underscore).
numfired.c	Return the number of rules fired so far. (Used for performance evaluation.)
p_compact.c	Compact the system.
p_split.c	Split an idle partition into a number of new partitions.
random.c	Random number generator. (Used for simulation purposes.)
rddg.c	Reduce the input data dependency graph.

teval.c	Evaluate the symbolic time equations from the database.
tics.c	Return the CPU time used so far in microseconds. (Used for performance evaluation.)
usrfuncs.c	Part of the CLIPS release. Modified to list all external C functions.

The following example run shows one sample execution of task II.

```
% clips
         CLIPS (V4.20 4/29/88)
CLIPS> (batch "disc")
CLIPS> (load "RULES/disc.init.rules")
*$****
CLIPS> (load "RULES/disc.load.rules")
*********
CLIPS> (load "RULES/disc.main1.rules")
**********************
CLIPS> (load "RULES/disc.main2.rules")
********
CLIPS> (load "RULES/disc.main3.rules")
****
CLIPS> (load "RULES/disc.main4.rules")
**********
CLIPS> (load "RULES/disc.cleanup.rules")
*******
CLIPS> (reset)
CLIPS> (dribble-on "run")
CLIPS> (run)
Enter the name of the data dependency graph file: ddg.2

Data dependency graph load complete.
Loading database object_recognition
Loading database median_filter
Loading database edge_detect
Loading database edge_link
Loading database edge_continuity
Loading database texture_analysis
Loading database boundary_trace
Loading database shape_analysis
Loading database region_formation
Loading the parameter file.
Convert parameter equations to numeric form.
Schedule algorithm median_filter in partition 0
Execute implementation median_filter_2 for node 0
-------------------------------------------------------
load is running for median_filter_2
    with time 0.0125
    and data INPUT_IMAGE I A O
Implementation median_filter_2 of node 0 is running
    in partition 0 of size 1024
    with data INPUT_IMAGE I A O
    and expected execution time of 18
Algorithm scheduling time: 25
Algorithm 0 is now finished.
Schedule algorithm edge_detect in partition 1
Execute implementation edge_detect_1 for node 2
-------------------------------------------------------
load is running for edge_detect_1
    with time 0.0125
```

```
    and data A I C O
Implementation edge_detect_1 of node 2 is running
    in partition 1 of size 512
    with data A I C O
    and expected execution time of 45
Algorithm scheduling time: 34
Schedule algorithm texture_analysis in partition 0
Execute implementation texture_analysis_1 for node 5
-----------------------------------------------------------
load is running for texture_analysis_1
    with time 0.0125
    and data A I D O
Implementation texture_analysis_1 of node 5 is running
    in partition 0 of size 512
    with data A I D O
    and expected execution time of 46
Algorithm scheduling time: 32
Algorithm 2 is now finished.
Schedule algorithm edge_link in partition 1
Execute implementation edge_link_1 for node 3
-----------------------------------------------------------
load is running for edge_link_1
    with time 0.0125
    and data _TANKS_FLIR I C M
Implementation edge_link_1 of node 3 is running
    in partition 1 of size 512
    with data _TANKS_FLIR I C M
    and expected execution time of 70
Algorithm scheduling time: 27
Algorithm 5 is now finished.
Algorithm 3 is now finished.
Schedule algorithm edge_continuity in partition 0
Execute implementation edge_continuity_1 for node 4
-----------------------------------------------------------
load is running for edge_continuity_1
    with time 0.0125
    and data C I _0.9 C
Implementation edge_continuity_1 of node 4 is running
    in partition 0 of size 1024
    with data C I _0.9 C
    and expected execution time of 25
Algorithm scheduling time: 26
Algorithm 4 is now finished.
Schedule algorithm edge_link in partition 0
Execute implementation edge_link_1 for node 3
-----------------------------------------------------------
load is running for edge_link_1
    with time 0.0125
    and data _TANKS_FLIR I C M
Implementation edge_link_1 of node 3 is running
    in partition 0 of size 1024
    with data _TANKS_FLIR I C M
    and expected execution time of 56
Algorithm scheduling time: 25
Algorithm 3 is now finished.
Schedule algorithm edge_continuity in partition 0
```

```
Execute implementation edge_continuity_1 for node 4
-----------------------------------------------------
convert_allocation is running for edge_continuity_1
    with time 3
    and data region row
Implementation edge_continuity_1 of node 4 is running
    in partition 0 of size 1024
    with data C I _0.9 C
    and expected execution time of 25
Algorithm scheduling time: 26
Algorithm 4 is now finished.
Schedule algorithm edge_link in partition 0
Execute implementation edge_link_1 for node 3
-----------------------------------------------------
load is running for edge_link_1
    with time 0.0125
    and data _TANKS_FLIR I C M
Implementation edge_link_1 of node 3 is running
    in partition 0 of size 1024
    with data _TANKS_FLIR I C M
    and expected execution time of 56
Algorithm scheduling time: 25
Algorithm 3 is now finished.
Schedule algorithm edge_continuity in partition 0
Execute implementation edge_continuity_1 for node 4
-----------------------------------------------------
convert_allocation is running for edge_continuity_1
    with time 3
    and data region row
Implementation edge_continuity_1 of node 4 is running
    in partition 0 of size 1024
    with data C I _0.9 C
    and expected execution time of 25
Algorithm scheduling time: 26
Algorithm 4 is now finished.
Schedule algorithm boundary_trace in partition 0
Execute implementation boundary_trace_1 for node 6
-----------------------------------------------------
load is running for boundary_trace_1
    with time 0.0125
    and data _TANKS_FLIR I C I D I E O
Implementation boundary_trace_1 of node 6 is running
    in partition 0 of size 1024
    with data _TANKS_FLIR I C I D I E O
    and expected execution time of 50
Algorithm scheduling time: 25
Algorithm 6 is now finished.
Schedule algorithm shape_analysis in partition 1
Execute implementation shape_analysis_1 for node 7
-----------------------------------------------------
load is running for shape_analysis_1
    with time 0.0125
    and data E I F O
Implementation shape_analysis_1 of node 7 is running
    in partition 1 of size 512
    with data E I F O
```

```
    and expected execution time of 262.58511353
Algorithm scheduling time: 28
Schedule algorithm region_formation in partition 0
Execute implementation region_formation_2 for node 8
---------------------------------------------------------
load is running for region_formation_2
    with time 0.0125
    and data _TANKS_FLIR I D I E I G O
Implementation region_formation_2 of node 8 is running
    in partition 0 of size 512
    with data _TANKS_FLIR I D I E I G O
    and expected execution time of 76
Algorithm scheduling time: 26
Algorithm 8 is now finished.
Algorithm 7 is now finished.
Schedule algorithm object_recognition in partition 0
Execute implementation object_recognition_1 for node 9
---------------------------------------------------------
load is running for object_recognition_1
    with time 0.0125
    and data F I G I
Implementation object_recognition_1 of node 9 is running
    in partition 0 of size 1024
    with data F I G I
    and expected execution time of 594.49395752
Algorithm scheduling time: 20
Algorithm 9 is now finished.

Enter the name of the DDG file ('exit' if done): exit

1165 rules fired
CLIPS> (dribble-off)
CLIPS> (exit)
```

C.3 Code Listings

The following listings of source code and example files are in alphabetic order by file name. The name of the file is in the upper right corner of each page.

assert_RDDG.c

```
/*
  assert_RDDG - assert the DDG nodes

  Input:
        infile - file name of reduced graph

  Output:
        1 - conversion successful
        0 - conversion error

  Side Effects:
        For each node, output is asserted to CLIPS as facts of the form:
        NODE <n> name <name>
        NODE <n> type <type>
        NODE <n> priority <priority>
        NODE <n> params <#_params param_1 param_type_1 ... param_n param_type_n>
        if node is "condition" type:
                NODE <n> descend num_descend rel_op num_T_nodes T_node_# ...
                                num_F_nodes F_node_# ...
        if node is "algorithm" type
                NODE <n> descend num_descendants num_children node_# ...

  Input File Format:
        n               {'n' is a unique integer >= 0}
        node_type priority      {"A" for an algorithm, "C" for a condition,
                                0 - 9 (9 is max)}
        name            {as listed in the DB}
        parameter list  {# of params, p1, p1type, ..., pn, pntype
                         p#type is "I", "M", "O" for input, mod input, output}
        if the node is a condition node
                then condition, number of TRUE nodes and list of TRUE nodes
                        number of FALSE nodes and list of FALSE nodes
                else number of children and list of child nodes
        number of descendants  {negative if in a loop}
*/

#include <stdio.h>
#include <string.h>
#include <math.h>
#include "clips.h"              /* CLIPS header for interface */

#define MAXNAME 30              /* max function name length */
#define MAXPARAMS 30            /* max number of parameters to function */
#define MAXCHILDREN 30          /* max number of children */
#define MAXNODES 100            /* max number of nodes in DDG */

struct NODE {                   /* node of the unreduced DDG */
      char type;                /* 'A' or 'C' */
      int priority;             /* 0 - 9, (9 is max) */
      char name[MAXNAME];       /* as listed in the DB */
      int numparams;            /* number of parameters */
      char param[MAXPARAMS][MAXNAME]; /* parameters */
      char ptype[MAXPARAMS];    /* 'C', 'I', 'M', 'O' */
      char relop[3];            /* relational operator for functions */
      int numtrue;              /* number of TRUE goto nodes */
      int tgoto[MAXCHILDREN];   /* node numbers if condition true */
      int numfalse;             /* number of FALSE nodes */
      int fgoto[MAXCHILDREN];   /* node numbers if condition false */
```

```
        int numchild;              /* number of child nodes */
        int children[MAXCHILDREN]; /* children of node */
        int numdesc;               /* number of descendants */
};

typedef struct NODE node;
node *nodelist[MAXNODES];          /* list of nodes */

/*********************************************************************/
int assert_RDDG (inname)
char *inname;
{
        FILE *infile;              /* input file pointer */
        int current;               /* current node location */
        int i;
        void newfact();            /* output a modified node to CLIPS */

        /* open the input file */
        if ((infile = fopen(inname,"r")) == NULL) {
                perror(inname);    /* error exit */
                return 0;
        }

        /* set each node to NULL (non-existent) */
        for (i = 0; i < MAXNODES; i++)
          nodelist[i] = (node *) 0;

        /* read in each node */
        while (!feof(infile)) {
                /* read the node number and allocate the node */
                fscanf(infile,"%d0,&current);
                nodelist[current] = (node *)malloc(sizeof(node));

                /* read the type, priority, and name */
                fscanf(infile,"%c%d187s",&(nodelist[current]->type),
                        &(nodelist[current]->priority), &(nodelist[current]->name[0]));

                /* read in the parameters */
                fscanf(infile,"%d",&(nodelist[current]->numparams));
                for (i = 0; i < nodelist[current]->numparams; i++)
                        fscanf(infile," %s %c",&(nodelist[current]->param[i][0]),
                                &(nodelist[current]->ptype[i]));

                /* the rest depends on the node type */
                if (nodelist[current]->type == 'C') {
                        /* conditional - read relop, tgoto, and fgoto */
                        fscanf(infile,"%s %d",&(nodelist[current]->relop[0]),
                                &(nodelist[current]->numtrue));
                        for (i = 0; i < abs(nodelist[current]->numtrue); i++)
                          fscanf(infile,"%d",&(nodelist[current]->tgoto[i]));
                        fscanf(infile,"%d", &(nodelist[current]->numfalse));
                        for (i = 0; i < abs(nodelist[current]->numfalse); i++)
                          fscanf(infile,"%d",&(nodelist[current]->fgoto[i]));
                }
                else {
                        /* algorithm - read children */
                        fscanf(infile,"%d",&(nodelist[current]->numchild));
                        for (i = 0; i < abs(nodelist[current]->numchild); i++)
```

```
                fscanf(infile,"%d",&(nodelist[current]->children[i]));
            }
            fscanf(infile,"%d",&(nodelist[current]->numdesc));
        }

        /* output each node */
        for (i = 0; i < MAXNODES; i++)
          if (nodelist[i] != (node *) 0)
            newfact(nodelist[i], i); /* output the resulting node */

        fclose(infile);           /* clean up and exit */
        return 1;
}

/*****************************************************************************
  newfact
        output a modified node to CLIPS as a fact

        INPUT: pnode - pointer to node to output
               n - node number

        SIDE EFFECTS:
                For each node, output is asserted to CLIPS as facts of the form:
                NODE <n> name <name>
                NODE <n> type <type>
                NODE <n> priority <priority>
                NODE <n> params <#_params param_1 param_type_1 ...>
                if node is "condition" type:
                        NODE <n> descend num_descend rel_op num_T_nodes
                                 T_node_# ... num_F_nodes F_node_# ...
                if node is "algorithm" type
                        NODE <n> descend num_descendants num_children node_# ...
*/
void newfact(pnode, n)
node *pnode;
int n;
{
        int i;
        char fact[256];           /* holds the fact when it is finished */
        char temp[80];            /* used for temp formatting storage */
        struct fact *assert();    /* function for asserting a fact to CLIPS */

        sprintf(fact, "NODE %d name %s", n, pnode->name);
        assert(fact);             /* put the fact into CLIPS */
        sprintf(fact, "NODE %d type %c", n, pnode->type);
        assert(fact);             /* put the fact into CLIPS */
        sprintf(fact, "NODE %d priority %d", n, pnode->priority);
        assert(fact);             /* put the fact into CLIPS */
        sprintf(fact, "NODE %d params %d ", n, pnode->numparams);

        /* add the parameters */
        for (i = 0; i < pnode->numparams; i++) {
              sprintf(temp, "%s %c ", pnode->param[i], pnode->ptype[i]);
              strcat(fact,temp);
        }
        assert(fact);             /* put the fact into CLIPS */

        sprintf(fact, "NODE %d descend %d ", n, pnode->numdesc);
```

```
        if (pnode->type == 'C') {
              sprintf(temp,"%s %d ", pnode->relop,pnode->numtrue);
              strcat(fact,temp);

              for (i = 0; i < abs(pnode->numtrue); i++) {
                    sprintf(temp, "%d ", pnode->tgoto[i]);
                    strcat(fact,temp);
              }

              sprintf(temp,"%d ",pnode->numfalse);
              strcat(fact,temp);

              for (i = 0; i < abs(pnode->numfalse); i++) {
                    sprintf(temp, "%d ", pnode->fgoto[i]);
                    strcat(fact,temp);
              }
        }
        else {
              sprintf(temp, "%d ",pnode->numchild);
              strcat(fact,temp);

              for (i = 0; i < abs(pnode->numchild); i++) {
                    sprintf(temp, "%d ", pnode->children[i]);
                    strcat(fact,temp);
              }
        }

        assert(fact);              /* put the fact into CLIPS */
}
```

db.region_formation

```
(DB region_formation var_meaning M pixels_per_row)
(DB region_formation region_formation_1)
(DB region_formation_1 mode s)
(DB region_formation_1 num_pe "<=" var M)
(DB region_formation_1 time 32 "(* (/ M 32) (+ 5 (* 2 (1 M 2))))")
(DB region_formation_1 time 64 "(* (/ M 64) (+ 5 (* 2 (1 M 2))))")
(DB region_formation_1 time 128 "(* (/ M 128) (+ 5 (* 2 (1 M 2))))")
(DB region_formation_1 time 256 "(* (/ M 256) (+ 5 (* 2 (1 M 2))))")
(DB region_formation_1 time 512 "(* (/ M 512) (+ 5 (* 2 (1 M 2))))")
(DB region_formation_1 time 1024 "(* (/ M 1024) (+ 5 (* 2 (1 M 2))))")
(DB region_formation_1 in_format byte byte byte)
(DB region_formation_1 out_format byte)
(DB region_formation_1 in_alloc region region region)
(DB region_formation_1 out_alloc region)
(DB region_formation region_formation_2)
(DB region_formation_2 mode m)
(DB region_formation_2 num_pe "<=" 1024)
(DB region_formation_2 time 32 "(* (/ M 32) (+ 5 (* 2 (1 M 2))))")
(DB region_formation_2 time 64 "(* (/ M 64) (+ 5 (* 2 (1 M 2))))")
(DB region_formation_2 time 128 "(* (/ M 128) (+ 5 (* 2 (1 M 2))))")
(DB region_formation_2 time 256 "(* (/ M 256) (+ 5 (* 2 (1 M 2))))")
(DB region_formation_2 time 512 "(* (/ M 512) (+ 5 (* 2 (1 M 2))))")
(DB region_formation_2 time 1024 "(* (/ M 1024) (+ 5 (* 2 (1 M 2))))")
(DB region_formation_2 in_format byte byte byte)
(DB region_formation_2 out_format byte)
(DB region_formation_2 in_alloc region region region)
(DB region_formation_2 out_alloc region)
```

db.template

```
;template:
;       (DB alg_name "var_meaning" variable meaning)
;       (DB alg_name imp_name)
;       (DB imp_name "mode" mode)
;       (DB imp_name "num_pe" #_PEs)
;       (DB imp_name "time 32" time)
;       (DB imp_name "time 64" time)
;       (DB imp_name "time 128" time)
;       (DB imp_name "time 256" time)
;       (DB imp_name "time 512" time)
;       (DB imp_name "time 1024" time)
;       (DB imp_name "in_format" input_format ...)
;       (DB imp_name "out_format" output_format ...)
;       (DB imp_name "in_alloc" input_allocation ...)
;       (DB imp_name "out_alloc" output_allocation ...)
;
;
;       mode: "s" | "m"
;       # PEs: ["<=" | ">="] # | <"var" variable_name>
;       time: <time for 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, or 1024
;             pixels as a function of image parameters> | "1" for unknown
;       input_format: <<"byte" | "integer" | "float" | "long" | "double">
;                     ["color"]> | string_type | "bit" | "selectable" | "none"
;       output_format: same as input format
;       input_allocation:  "none" | "row" | "column" | "region" | other
;       output_allocation: same a input allocation
```

ddg.1

```
0
A 0
median_filter
2 INPUT_IMAGE I A O
3 1 2 3
1
A 0
edge_detect
2 A I B O
1 4
2
A 0
edge_detect
2 A I C O
1 4
3
A 0
edge_detect
2 A I D O
1 4
4
A 0
region_formation
4 B I C I D I X O
0
```

ddg.2

```
0
A 0
median_filter
2 INPUT_IMAGE I A O
2 2 5
2
A 0
edge_detect
2 A I C O
1 3
3
A 0
edge_link
2 _TANKS_FLIR I C M
1 4
4
C 0
edge_continuity
2 C I _0.9 C
<= 1 3 1 6
5
A 0
texture_analysis
2 A I D O
2 6 8
6
A 0
boundary_trace
4 _TANKS_FLIR I C I D I E O
2 7 8
7
A 0
shape_analysis
2 E I F O
1 9
8
A 0
region_formation
4 _TANKS_FLIR I D I E I G O
1 9
9
A 0
object_recognition
2 F I G I
0
```

ddg.3

```
0
A 0
shape_analysis
2 INPUT_IMAGE I A O
0
```

ddg.4

```
0
A 0
median_filter
2 INPUT_IMAGE I C O
3 4 5 6
1
A 1
object_recognition
3 INPUT_IMAGE I INPUT_IMAGE I A O
1 8
2
A 0
median_filter
2 INPUT_IMAGE I D O
3 4 5 6
3
A 1
object_recognition
3 INPUT_IMAGE I INPUT_IMAGE I B O
1 8
4
A 0
boundary_trace
3 C I D I E O
1 7
5
A 0
boundary_trace
3 C I D I F O
1 7
6
A 0
boundary_trace
3 C I D I G O
1 7
7
A 0
region_formation
4 E I F I G I H O
1 8
8
A 0
region_formation
4 A I B I H I X O
0
```

ddg.5

```
0
A 0
median_filter
2 INPUT_IMAGE I A O
0
1
A 0
median_filter
2 INPUT_IMAGE I A O
0
2
A 0
median_filter
2 INPUT_IMAGE I A O
0
3
A 0
median_filter
2 INPUT_IMAGE I A O
0
4
A 0
median_filter
2 INPUT_IMAGE I A O
0
5
A 0
median_filter
2 INPUT_IMAGE I A O
0
6
A 0
median_filter
2 INPUT_IMAGE I A O
0
7
A 0
median_filter
2 INPUT_IMAGE I A O
0
8
A 0
median_filter
2 INPUT_IMAGE I A O
0
9
A 0
median_filter
2 INPUT_IMAGE I A O
0
10
A 0
median_filter
2 INPUT_IMAGE I A O
0
11
A 0
median_filter
```

ddg.5

```
2 INPUT_IMAGE I A O
0
12
A 0
median_filter
2 INPUT_IMAGE I A O
0
13
A 0
median_filter
2 INPUT_IMAGE I A O
0
14
A 0
median_filter
2 INPUT_IMAGE I A O
0
15
A 0
median_filter
2 INPUT_IMAGE I A O
0
```

```
(load "RULES/disc.init.rules")
(load "RULES/disc.load.rules")
(load "RULES/disc.main1.rules")
(load "RULES/disc.main2.rules")
(load "RULES/disc.main3.rules")
(load "RULES/disc.main4.rules")
(load "RULES/disc.cleanup.rules")
(reset)
(dribble-on "run")
```

disc.cleanup.rules

```
; DISC
;       Dynamic Intelligent Scheduling and Control
;               Clean Up Section
;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;       Clean Up Section
;       ----------------
;
;       A) Reset the machine to the initial state (1024 x 1).
;       B) Reset fact list.
;       C) Indicate that system is ready for the new job.
;
;
;       1000 <= salience <= 1999
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defrule reset_1                        ;remove old partition definitions
        (declare (salience 1900))
        ?fact1 <- (PARTITION ?state ?size ?num)
=>
        (if (or (neq ?state I) (neq ?size 1024) (neq ?num 0))
            then
                (retract ?fact1)
        )
)

(defrule reset_2                        ;remove old partition content definitions
        (declare (salience 1900))
        ?fact1 <- (Partition $?)
=>
        (retract ?fact1)
)

(defrule reset_6                        ;remove old time evaluations
        (declare (salience 1900))
        ?fact1 <- (DB ? etime $?)
=>
        (retract ?fact1)
)

(defrule reset_3                        ;remove old node definitions
        (declare (salience 1900))
        ?fact1 <- (NODE $?)
=>
        (retract ?fact1)
)

(defrule reset_4                        ;remove old definitions - add some new
        (declare (salience 1800))
        ?fact1 <- (SYS_TIME ?)
        ?fact2 <- (FEX ? $?)
        ?fact3 <- (NUM_IP ?)
=>
        (retract ?fact1 ?fact2 ?fact3)
        (assert (PARTITION I 1024 0))   ;initial system state
        (assert (NUM_IP 1))
```

```
        (assert (FEX))
        (assert (SYS_TIME 0))
)

(defrule reset_5                        ;remove old timer info
        (declare (salience 1900))
        ?fact1 <- (TIMER ?)
=>
        (retract ?fact1)
)

(defrule restart                        ;get things going again
        (declare (salience 1200))
        (NUM_PEN 0)
=>
        (assert (STARTUP))              ;indicate system reset
        (fprintout t crlf "Enter the name of the DDG file ('exit' if done): ")
        (bind ?ddg (read))              ;get the file name
        (fprintout t crlf)
        (if (eq ?ddg exit) then (halt)) ;done - exit
        (bind ?results (initialize ?ddg))       ;init the DDG
        (if (= ?results 0)              ;see if error occurred
           then
               (fprintout t "Error in data dependency graph processing"
                       " of file: " ?ddg crlf
                        "Execution has been halted" crlf)
               (halt)
           else
               (fprintout t "Data dependency graph load complete." crlf)
         )
)
```

disc.init.rules

```
; DISC
;       Dynamic Intelligent Scheduling and Control
;               Initialization Section
;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;       INITIALIZATION RULES
;       --------------------
;
;
; These rules set up CLIPS by reading in the RDDG (the file name is
; prompted for) and showing the system in its default state (M MC's
; with P PE's each).
;
; For each RDDG node, there are facts of the form:
; NODE <n> name <name>
; NODE <n> type <type>
; NODE <n> priority <priority>
; NODE <n> params <#_params param_1 param_type_1 ... param_n param_type_n>
; if node is "condition" type:
;       NODE <n> descend num_descend rel_op num_T_nodes T_node_# ...
;                         num_F_nodes F_node_# ...
; if node is "algorithm" type
;       NODE <n> descend num_descendants num_children node_# ...
;
;       9900 < salience <= 10000
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defrule initialize                        ;read the RDDG
        (declare (salience 10000))         ;this rule fires first
        ?fact1 <- (initial-fact)           ;initial fact forces firing
=>
        (retract ?fact1)
        (assert (STARTUP))                 ;indicate system initialization
        (assert (DB_LOADED))               ;no DBs loaded now
        (fprintout t "Enter the name of the data dependency graph file: ")
        (bind ?ddg (read))                 ;get the file name
        (fprintout t crlf)
        (bind ?results (initialize ?ddg))         ;init the DDG
        (if (= ?results 0)                 ;see if error occurred
            then
                (fprintout t "Error in data dependency graph processing"
                        " of file: " ?ddg crlf
                         "Execution has been halted" crlf)
                (halt)
            else
                (fprintout t "Data dependency graph load complete." crlf)
                (load-facts "DB/db.conversion") ;load the conversion db
         )
)

;These facts show the machine in its initial state of 32 MCs
;with 32 PEs each.

(deffacts initial_state
        (DONE_TIME)                        ;list of algorithm finish times
        (SYS_TIME 0)                       ;internal timer
```

```
        (NUM_PEN 0)                         ;number of PEN algorithms
        (NUM_IP 1)                          ;number of idle partitions
        (FEX)                               ;null Finished EXecuting list
        (MACHINE_INFO MC 32 PE 32)          ;machine size
        (MACHINE_INFO MST 0)                ;negligible mode switch time
        (P_MERGEABLE 64 0 16)               ;64 PE partitions
        (P_MERGEABLE 64 1 17)
        (P_MERGEABLE 64 2 18)
        (P_MERGEABLE 64 3 19)
        (P_MERGEABLE 64 4 20)
        (P_MERGEABLE 64 5 21)
        (P_MERGEABLE 64 6 22)
        (P_MERGEABLE 64 7 23)
        (P_MERGEABLE 64 8 24)
        (P_MERGEABLE 64 9 25)
        (P_MERGEABLE 64 10 26)
        (P_MERGEABLE 64 11 27)
        (P_MERGEABLE 64 12 28)
        (P_MERGEABLE 64 13 29)
        (P_MERGEABLE 64 14 30)
        (P_MERGEABLE 64 15 31)
        (P_MERGEABLE 128 0 8)               ;128 PE partitions
        (P_MERGEABLE 128 1 9)
        (P_MERGEABLE 128 2 10)
        (P_MERGEABLE 128 3 11)
        (P_MERGEABLE 128 4 12)
        (P_MERGEABLE 128 5 13)
        (P_MERGEABLE 128 6 14)
        (P_MERGEABLE 128 7 15)
        (P_MERGEABLE 256 0 4)               ;256 PE partitions
        (P_MERGEABLE 256 1 5)
        (P_MERGEABLE 256 2 6)
        (P_MERGEABLE 256 3 7)
        (P_MERGEABLE 512 0 2)               ;512 PE partitions
        (P_MERGEABLE 512 1 3)
        (P_MERGEABLE 1024 0 1)              ;1024 PE partition
        (PARTITION I 1024 0)                ;initial state
)

;this rule builds the initial "Potentially Executable Now" list by looking
;for NODES with inputs only of "INPUT_IMAGE", a constant, or NODES with no inputs

(defrule build_PEN_list_1
        (declare (salience 9950))
        (STARTUP)
        ?fact1 <- (NUM_PEN ?npen)                   ;current number of PEN algorithms
        (NODE ?nnum params ?nparam $?rest)          ;node to possibly add
        (not (PEN ?nnum))                           ;not already added
  =>
        (bind ?np ?nparam)                          ;need a variable for looping
        (bind ?flag 1)                              ;default to "add this node"
        (while (> ?np 0)                            ;look at each parameter
                (if (&& (neq 0 (nth (* 2 ?np) $?rest))  ;output
                        (neq INPUT_IMAGE (nth (- (* 2 ?np) 1) $?rest)) ;INPUT_IMAGE
                        (neq _ (letter (nth (- (* 2 ?np) 1) $?rest)))) ;constant
                    then
                        (bind ?flag 0)
                )
```

```
                (bind ?np (- ?np 1))
        )
        (if (= ?flag 1)                          ;add alg to list
            then
                (retract ?fact1)
                (assert (NUM_PEN =(+ 1 ?npen)))  ;bump PEN count by 1
                (assert (PEN ?nnum))
        )
)

;These rules build up a list of the parents of each node.

(defrule find_parents_a                          ;algorithm nodes
        (declare (salience 9940))
        (STARTUP)
        (NODE ?nnum type A)                      ;node to work on
        (NODE ?nnum name ?)
        (NODE ?nnum descend ? ?cnum $?clist)     ;children
=>
        (assert (NODE ?nnum parents))            ;null list
        (bind ?num ?cnum)                        ;loop variable
        (while (> ?num 0)
                (assert (PARENT ?nnum =(nth ?num $?clist)))
                (bind ?num (- ?num 1))
        )
)

(defrule find_parents_c                          ;condition nodes
        (declare (salience 9940))
        (STARTUP)
        (NODE ?nnum type C)                      ;node to work on
        (NODE ?nnum name ?)
        (NODE ?nnum descend ? ? ?tnum $?clist)   ;children
=>
        (assert (NODE ?nnum parents))            ;null list
        (bind ?num ?tnum)                        ;loop variable
        (while (> ?num 0)                        ;TRUE children
                (assert (PARENT ?nnum =(nth ?num $?clist)))
                (bind ?num (- ?num 1))
        )
        (bind ?start (+ 1 (abs ?tnum)))          ;start of FALSE list
        (bind ?num (nth ?start $?clist))         ;FALSE list length
        (while (> ?num 0)                        ;FALSE children
                (assert (PARENT ?nnum =(nth (+ ?num ?start) $?clist)))
                (bind ?num (- ?num 1))
        )
)

(defrule find_parents_build
        (declare (salience 9940))
        (STARTUP)
        ?fact1 <- (NODE ?nnum parents $?plist)   ;parent list so far
        ?fact2 <- (PARENT ?parent ?nnum)         ;parent of this node
=>
        (retract ?fact1 ?fact2)
        (assert (NODE ?nnum parents $?plist ?parent))   ;new parent list
)
```

disc.load.rules

```
; DISC
;        Dynamic Intelligent Scheduling and Control
;                Load Section
;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;        LOAD RULES
;        ----------
;
; These rules load in the facts contained in the named database.
;
;        9700 < salience <= 9900
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defrule fact_load                           ;do actual fact loading
        (declare (salience 9890))            ;this rule must fire after RDDG load
        (STARTUP)
        (NODE ? name ?name)                  ;database to load
        ?fact1 <- (DB_LOADED $?dblist)       ;databases already loaded
=>
        (if (eq (member ?name $?dblist) 0)
            then                             ;not already loaded
                (bind ?dbname (str_cat "DB/db." ?name))
                (fprintout t "Loading database " ?name crlf)
                (load-facts ?dbname)
                (retract ?fact1)
                (assert (DB_LOADED $?dblist ?name))      ;mark as loaded
        )
)

(defrule parameter_load                      ;load in the parameter values
        (declare (salience 9850))
        (STARTUP)                            ;only here because a rule must have an LHS
=>
        (fprintout t "Loading the parameter file." crlf)
        (load-facts "task.parameters")  ;parameter value file
        (fprintout t "Convert parameter equations to numeric form." crlf)
)

; convert the algorithm time from equation form to number form

(defrule get_variable_values                 ;get a value for each variable
        (declare (salience 9800))            ;must get values prior to evaluation
        (STARTUP)
        (DB ?class var_meaning ?var ?meaning)
        (Parameter ?meaning ?val)
=>
        (assert (DB ?class var_val ?var ?val))
        (assert (DB ?class var_list))
)

(defrule remove_parameters                   ;clean up unnecessary parameter values
        (declare (salience 9795))            ;must come AFTER they are used
        (STARTUP)
        ?fact1 <- (Parameter $?)
=>
```

```
        (retract ?fact1)
)

(defrule make_value_list                ;make a list of variable-value pairs
        (declare (salience 9790))
        (STARTUP)
        ?fact1 <- (DB ?class var_val ?var ?val)
        ?fact2 <- (DB ?class var_list $?list)
=>
        (assert (DB ?class var_list $?list ?var ?val))
        (retract ?fact1 ?fact2)
)

(defrule eval_times                     ;evaluate the time equations
        (declare (salience 9780))
        (STARTUP)
        (DB ?class var_list $?varval)
        (DB ?class ?name)
        (DB ?name time ?tnpe ?time)
=>
        (assert (DB ?name etime ?tnpe =(teval ?time $?varval)))
)

(defrule eval_numpe                     ;evaluate number of PEs bound
        (declare (salience 9770))
        (STARTUP)
        ?fact <- (DB ?name num_pe $?le var ?v)
        (DB ?class ?name)
        (DB ?class var_list $? ?v ?val $?)
=>
        (assert (DB ?name num_pe $?le ?val))
        (retract ?fact)
)

(defrule clean_var_list                 ;retract the variable value list
        (declare (salience 9710))
        (STARTUP)
        ?fact1 <- (DB ? var_list $?)
=>
        (retract ?fact1)
)

(defrule startup_done                   ;indicate scheduling can begin
        (declare (salience 9700))
        ?fact1 <- (STARTUP)             ;no more startup
=>
        (retract ?fact1)
        (assert (Partition 0 s NONE X X))       ;initial configuration
        ;template: (Partition # <mode> <data> <data allocation> <data format>)
        ;initial: 1024 PE partition, SIMD, no data allocated
        (assert (schedule next))        ;partition 0 is initially idle
        (assert (TIMER =(numfired)))    ;timing function
)
```

disc.main1.rules

```
; DISC
;       Dynamic Intelligent Scheduling and Control
;               Main Loop Section 1
;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;       MAIN DISC LOOP
;       --------------
;
;       I)    Look for "schedule next" fact and then
;             1) deal with repartitioning based on:
;               A) number of free partitions
;               B) number of PEN algorithms
;
;             2) choose which algorithm will go next based on:
;               A) # of descendants  (Use Loop_weight * #_descendants for loops.)
;               B) Correct data already in some partition
;               C) Priority value - this is the overriding condition
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;       Algorithm Selection Rules
;       (DISC step I)
;
;       These rules select which of the PEN list algorithms
;       should be executed next now that a partition is free.
;
;       8000 <= salience <= 8999

;if there is just one free partition and 1 PEN algorithm, then just run it
;
(defrule repartition_1                   ;1 PEN alg, 1 free partition
        (declare (salience 8950))
        ?fact1 <- (schedule next)
        (NUM_PEN 1)                      ;1 PEN alg
        (NUM_IP 1)                       ;1 idle partition
        (PARTITION I ? ?pnum)            ;idle partition
=>
        (retract ?fact1)
        (assert (schedule next ?pnum))
        (assert (PENMP max DUMMY -1 x)) ;set up max vote finder
        (assert (PENP -1))               ;set up max priority finder
)

;if there is just one free partition and n (n > 1) PEN algorithms,
;and the partition is already small as possible, then
;use the partition for the next algorithm
;
(defrule repartition_2                   ;more than 1 PEN alg, 1 free partition
        (declare (salience 8950))
        ?fact1 <- (schedule next)
        (NUM_PEN ?numpen&:(> ?numpen 1))        ;more than 1 PEN alg
        (NUM_IP 1)                       ;1 free partition
        (MACHINE_INFO ? ? PE ?minpe)     ;minimum PEs per partition
        ?fact2 <- (PARTITION I ?minpe ?pnum $?plist)    ;idle partition
=>
```

```
        (retract ?fact1 ?fact2)
        (assert (schedule next ?pnum))
        (assert (PENMP max DUMMY -1 x)) ;set up max vote finder
        (assert (PENP -1))              ;set up max priority finder
)

;if there is just one free partition and n (n > 1) PEN algorithms,
;then split the partition into m (m = min(n, #_MC_in_partition))
;partitions and use the largest new partition for the next algorithm
;
(defrule repartition_3                  ;more than 1 PEN alg, 1 free partition
        (declare (salience 8949))
        ?fact1 <- (schedule next)
        (NUM_PEN ?numpen&:(> ?numpen 1))        ;more than 1 PEN alg
        ?fact4 <- (NUM_IP 1)            ;1 idle partition
        ?fact2 <- (PARTITION I ?psize ?pnum)    ;idle partition
        ?fact3 <- (Partition ?pnum $?)  ;partition contents
=>
        (retract ?fact1)
        (if (> ?psize 32)               ;partition can be split
            then
                (retract ?fact2 ?fact3 ?fact4)
                (assert (NUM_IP =(p_split ?pnum ?numpen ?psize)))
                (assert (PENPART x -1))         ;set up partition finder
        )
        (assert (PENMP max DUMMY -1 x)) ;set up max vote finder
        (assert (PENP -1))              ;set up max priority finder
)

;if there are p (p > 1) free partitions and only 1 PEN algorithm,
;then merge the partitions if they are adjacent
;
(defrule repartition_4                  ;1 PEN alg, more than 1 free partition
        (declare (salience 8950))
        (schedule next)
        (NUM_PEN 1)                     ;1 PEN algorithm
        ?fact6 <- (NUM_IP ?numip&:(> ?numip 1)) ;more than 1 free partition
        ?fact1 <- (PARTITION I ?psize ?pnum1)   ;idle partition 1
        ?fact2 <- (PARTITION I ?psize ?pnum2&:(neq ?pnum1 ?pnum2)) ;idle 2
        (P_MERGEABLE =(+ ?psize ?psize) ?pnum1 ?pnum2)  ;mergeable
        ?fact4 <- (Partition ?pnum1 $?) ;partition contents
        ?fact5 <- (Partition ?pnum2 $?)
=>
        (retract ?fact1 ?fact2 ?fact4 ?fact5 ?fact6)
        (assert (PARTITION I =(+ ?psize ?psize) ?pnum1))
        (assert (Partition ?pnum1 s NONE X X))  ;new partition contents
        (assert (NUM_IP =(- ?numip 1)))
        ;(p_merge ?pnum1 ?pnum2)        ;add system merge call for working OS
)

;if there are p (p > 1) free partitions and only 1 PEN algorithm
;and the partitions are not adjacent, then run the algorithm
;in the best partition
;
(defrule repartition_5                  ;1 PEN alg, more than 1 free partition
        (declare (salience 8940))
        ?fact1 <- (schedule next)
        (NUM_PEN 1)                     ;1 PEN alg
```

```
        (NUM_IP ?numip&:(> ?numip 1))    ;more than 1 free partition
=>
        (retract ?fact1)
        (assert (PENPART x -1))          ;set up partition finder
        (assert (PENMP max DUMMY -1 x)) ;set up max vote finder
        (assert (PENP -1))               ;set up max priority finder
)

;if there are p (P > 1) free partitions and n (n > 1) PEN algorithms,
;then if p == n use the best partition, else compact and use the
;largest partition
;
(defrule repartition_6                   ;more than 1 PEN alg, more than 1 partition
        (declare (salience 8950))
        ?fact1 <- (schedule next)
        (NUM_PEN ?numpen&:(> ?numpen 1))        ;more than 1 PEN alg
        ?fact2 <- (NUM_IP ?numip&:(> ?numip 1)) ;more than 1 free partition
=>
        (retract ?fact1)
        (if (neq ?numpen ?numip)
            then                         ;compact the system
                (retract ?fact2)         ;used in production system
                (assert (DUMP_SYS_INFO))        ;not needed in production system
                (open "_SYS_INFO_" sys_info "w")        ;not needed in production
                ;(assert (NUM_IP =(p_compact ?numpen))) used in production system
        )
        (assert (PENPART x -1))          ;set up partition finder
        (assert (PENMP max DUMMY -1 x)) ;set up max vote finder
        (assert (PENP -1))               ;set up max priority finder
)

;The system state dump is here because the test system does not
;know about the state of the parallel processor the way the real
;low-level OS would.  The retraction of the system state, however,
;must be done because p_compact asserts the new state.
;
(defrule repartition_6_2                 ;dump the system state info
        (declare (salience 8948))
        (DUMP_SYS_INFO)
        ?fact1 <- (PARTITION A ?size ?part)     ;active partition definition
=>
        (retract ?fact1)
        (fprintout sys_info ?size " " ?part crlf)
)

(defrule repartition_6_3                 ;retract the rest of the state info
        (declare (salience 8946))
        (DUMP_SYS_INFO)
        ?fact1 <- (PARTITION I ? ?part)
        ?fact2 <- (Partition ?part $?)
=>
        (retract ?fact1 ?fact2)
)

(defrule repartition_6_4                 ;compact the system
        (declare (salience 8940))
        ?fact1 <- (DUMP_SYS_INFO)
        (NUM_PEN ?numpen)
```

```
=>
        (retract ?fact1)
        (close sys_info)
        (assert (NUM_IP =(p_compact ?numpen)))
        (system "rm _SYS_INFO_")
)

(defrule repartition_8                  ;show partition as moved
        (declare (salience 8930))
        ?fact1 <- (P_CMOVE ?old ?new)   ;where old partition moved to
        ?fact2 <- (ACE ?node ?old)      ;ACE definition
        ?fact3 <- (Partition  ?old $?contents)  ;old partition contents
=>
        (retract ?fact1 ?fact2 ?fact3)
        (assert (NEW_PARTITION ?old ?new $?contents))   ;new location
        (assert (NEW_ACE ?node ?new))
        (fprintout t "COMPACT: move " ?old " to " ?new crlf)
)

(defrule repartition_8_2                ;clean the partition content lists
        (declare (salience 8920))
        ?fact1 <- (NEW_PARTITION ?old ?new $?contents)  ;real contents
        ?fact2 <- (NEW_ACE ?node ?new)  ;old location
=>
        (retract ?fact1 ?fact2)
        (assert (Partition ?new $?contents))    ;correct definition
        (assert (ACE ?node ?new))       ;new location
)

(defrule repartition_8_3                ;add any needed partition content defs
        (declare (salience 8910))
        (PEN ?)
        (PARTITION I ? ?pnum)           ;idle partition
        (not (Partition ?pnum $?))      ;no content definition
=>
        (assert (Partition ?pnum s none none none))
)

;now start to work on the algorithm selection
;
(defrule PEN_max_priority               ;find max priority of PEN algs
        (declare (salience 8890))
        ?fact1 <- (PENP ?maxp)          ;max priority so far
        (PEN ?nnum)                     ;alg can be executed
        (NODE ?nnum priority ?prior&:(> ?prior ?maxp))
                                        ;new alg has higher priority
=>
        (retract ?fact1)
        (assert (PENP ?prior))          ;assert new maximum
)

(defrule build_PENMP_list               ;build list of max priority nodes
        (declare (salience 8880))
        (PENP ?maxp)                    ;max priority
        (PEN ?nnum)                     ;node to check
        (NODE ?nnum priority ?maxp)     ;max priority node
=>
        (assert (PENMP ?nnum))          ;vote on this alg
```

```
)

(defrule clean_PENP                          ;clear unneeded max PENP
        (declare (salience 8840))
        ?fact1 <- (PENP ?)
=>
        (retract ?fact1)
)

(defrule PEN_choose_alg_a                    ;vote using # of descendants
        (declare (salience 8800))
        ?fact1 <- (PENMP ?nnum)              ;algorithm to consider
        (NODE ?nnum descend ?desc $?)        ;number of descendants
        (Loop_weight ?lweight)               ;loop weight factor
=>
        (retract ?fact1)                     ;no longer need priority list element
;       (fprintout t "Algorithm " ?nnum " has a descendant weight of ")
        (if (< ?desc 0)                      ;check for loop node
            then
                (assert (PENMP ?nnum A =(* ?lweight ?desc)))    ;loop node
;               (fprintout t (* ?lweight ?desc) crlf)
            else
                (assert (PENMP ?nnum A ?desc))  ;non-loop node
;               (fprintout t ?desc crlf)
        )
)

(defrule PEN_choose_alg_b                    ;vote using data matching
        (declare (salience 8800))
        ?fact1 <- (PENMP ?nnum A ?score)     ;algorithm to consider
        (Partition ?part ? ?pdata $?)        ;the data in the partition
        (NODE ?nnum params $?list)           ;node to consider
=>
        (bind ?where (member ?pdata $?list))    ;see if data is needed
        (if (&& (neq (nth (+ ?where 1) $?list) 0)       ;make sure it's an input
                (> ?where 0))                ;and it matches
            then
;               (fprintout t "Algorithm " ?nnum
;                            " uses the same data as partition " ?part crlf)
                (assert (PENMP ?nnum B =(+ 1 ?score) ?part))    ;new score
        )
)

(defrule PEN_choose_alg_c                    ;set default for data match vote
        (declare (salience 8750))
        ?fact1 <- (PENMP ?nnum A ?score)     ;old score
=>
        (retract ?fact1)
        (assert (PENMP ?nnum B ?score -1))   ;same score - no partition match
)

(defrule PEN_clean_A                         ;remove unneeded scores
        (declare (salience 8740))
        ?fact1 <- (PENMP ? A ?)              ;old score
=>
        (retract ?fact1)
)
```

disc.main1.rules

```
(defrule PEN_choose_alg_max                          ;find the maximum algorithm score
        (declare (salience 8700))
        ?fact1 <- (PENMP max ? ?maxval ?)            ;max score so far
        ?fact2 <- (PENMP ?nnum B ?score ?partnum)        ;score for this algorithm
=>
        (retract ?fact2)
        (if (> ?score ?maxval)                       ;new maximum
            then
                (retract ?fact1)                     ;dump old maximum
                (assert (PENMP max ?nnum ?score ?partnum))
        )
)

(defrule PEN_choose_partition                        ;find best free partition to use
        (declare (salience 8510))
        ?fact1 <- (PENPART ? ?score)                 ;current max score
        (Partition ?part $?)                         ;candidate free partition
        (PARTITION I ?psize&:(> ?psize ?score) ?part)
        (PENMP max ? ? ?match)                       ;data match partition
=>
        (retract ?fact1)
        (assert (PENPART ?part =(+ ?psize (eq ?match ?part))))
)

(defrule PEN_choose_partition_final                  ;assert best partition choice
        (declare (salience 8500))
        ?fact1 <- (PENPART ?usepart ?)
=>
        (retract ?fact1)
        (assert (schedule next ?usepart))            ;partition to use
)

(defrule PEN_choose_alg_final                        ;assert the final alg choice
        (declare (salience 8100))
        ?fact1 <- (PENMP max ?nnum ? ?partnum)       ;highest scoring algorithm
        (NODE ?nnum name ?name)                      ;alg name
        (schedule next ?usepart)                     ;partition choice
=>
        (retract ?fact1)
        (assert (execute algorithm ?nnum))           ;choose imp flag
        (fprintout t "Schedule algorithm " ?name " in partition " ?usepart crlf)
)
```

disc.main2.rules

```
; DISC
;       Dynamic Intelligent Scheduling and Control
;               Main Loop Section 2
;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;       MAIN DISC LOOP
;       --------------
;
;       II)  Decide which implementation of the selected algorithm to use.
;               Vote for the implementation based on (weights in ()'s ):
;               A) mode vs switch time (switch time)
;               B) #PEs required vs #PEs free (none, but rejection possible)
;               C) execution time (TIME[MIN PE] / TIME[N PE])
;               D) Data format vs partition data format (-conversion_time)
;               E) Data allocation vs partition allocation (-conversion_time)
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;       Implementation Selection Rules
;       (DISC step II)
;
;       These rules select which of the implementations of the
;       selected algorithm should be used.
;
;       7000 <= salience <= 7999

(defrule choose_implementation_init             ;set up voting scheme
        (declare (salience 7900))
        (execute algorithm ?)
=>
        (assert (implementation max DUMMY -1 -1e38))    ;dummy max score
)

;Vote based on the imp mode and the mode switch time
(defrule choose_implementation_a        ;find possible implementations
        (declare (salience 7900))
        (execute algorithm ?nnum)       ;algorithm to schedule
        (NODE ?nnum name ?name)         ;alg name
        (DB ?name ?iname)               ;candidate implementation
        (DB ?iname mode ?mode)          ;implementation mode
        (MACHINE_INFO MST ?stime)       ;machine mode switch time
        (schedule next ?partnum)        ;partition being considered
        (Partition ?partnum ?pmode $?)  ;partition mode
=>
;       (fprintout t "     Implementation " ?iname " mode switch vote: ")
        (if (eq ?mode ?pmode)           ;compare modes
            then                        ;same mode - full vote
                (assert (implementation ?iname ?nnum A ?stime))
;               (fprintout t ?stime crlf)
            else                        ;different mode - zero vote
                (assert (implementation ?iname ?nnum A 0))
;               (fprintout t 0 crlf)
        )
)
```

disc.main2.rules

```
;Vote based on the imp #PEs required vs. # PEs available.
;(Actually, no weight is given here, but the imp can be rejected.)
(defrule choose_implementation_b        ;find possible implementations
        (declare (salience 7900))
        ?fact1 <- (implementation ?name ?nnum A ?score) ;imp to schedule
        (DB ?name num_pe ?relop ?num)   ;algorithm #PEs required
        (schedule next ?part)           ;idle partition number
        (PARTITION I ?psize ?part)      ;idle partition
=>
        (retract ?fact1)
        (if (|| (&& (eq ?relop ">=")    ;see if partition meets requirement
                    (>= ?psize ?num))
                (&& (eq ?relop "<=")
                    (<= ?psize ?num)))
            then
                (assert (implementation ?name ?nnum B ?score))  ;OK to use
            else
                (assert (implementation ?name ?nnum B -1e7))    ;reject
;               (fprintout t "     Implementation " ?name
;                            " will not fit into the partition - rejected." crlf)
        )
)

;Vote based on the execution time for the #PEs available
(defrule choose_implementation_c        ;find possible implementations
        (declare (salience 7900))
        ?fact1 <- (implementation ?name ?nnum B ?score) ;imp to schedule
        (schedule next ?part)           ;idle partition number
        (PARTITION I ?psize ?part)      ;idle partition
        (DB ?name etime ?psize ?time)   ;algorithm #PEs required
        (MACHINE_INFO ? ? PE ?minpe)    ;min partition size
        (DB ?name etime ?minpe ?maxtime)        ;exec time for min PE
=>
        (retract ?fact1)
        ;vote based on time-weighted exec time
        (assert (implementation ?name ?nnum C =(+ ?score (/ ?maxtime ?time))))
;       (fprintout t "     Implementation " ?name " execution time vote: "
;                    (/ ?maxtime ?time) crlf)
)

;Vote based on the correct data and format and allocation in the partition
(defrule choose_implementation_d                ;find possible implementations
        (declare (salience 7900))
        ?fact1 <- (implementation ?name ?nnum C ?score) ;imp to schedule
        (NODE ?nnum params $?nlist)             ;alg inputs
        (schedule next ?part)                   ;partition to examine
        (Partition ?part ? ?pdata ?palloc ?pformat)     ;partition data
        (PARTITION I ?psize ?part)              ;idle partition
        (DB convert_format etime ?psize ?ftime) ;format conversion time
        (DB convert_allocation etime ?psize ?atime)     ;allocation conversion time
        (DB ?name in_alloc $?ialloc)            ;imp input allocation
        (DB ?name in_format $?iformat)          ;imp input format
        (Load_constant ?loadc)                  ;load time multiplier constant
=>
        (retract ?fact1)
;       (fprintout t "     Implementation " ?name
;            " data format and allocation score:" crlf)
        (if (|| (> (length $?iformat) 1)                ;more than 1 input
```

```
            (= 0 (member ?pdata $?nlist)))          ;incorrect data
        then                                        ;load - subtract load time
            (assert (implementation ?name ?nnum E
                    =(- ?score (* ?loadc (+ ?atime ?ftime)))))
            (assert (Do load ?name =(* ?loadc (+ ?ftime ?atime))
                    =(mv-subseq 2 (length $?nlist) $?nlist)))
;           (fprintout t "        load: "
;                   (- 0 (* ?loadc (+ ?ftime ?atime))) crlf)
        else                                        ;do conversions
            (bind ?vote 0)
            (if (neq ?pformat (nth 1 $?iformat))    ;reformat the data
                then
                    (bind ?vote (- ?vote ?ftime))
                    (assert (Do convert_format ?name ?ftime ?pformat
                            =(nth 1 $?iformat)))
;                   (fprintout t "        format: " (- 0 ?ftime) crlf)
            )
            (if (neq ?palloc (nth 1 $?ialloc))      ;reallocate the data
                then
                    (bind ?vote (- ?vote ?atime))
                    (assert (Do convert_allocation ?name ?atime ?palloc
                            =(nth 1 $?ialloc)))
;                   (fprintout t "        allocation: " (- 0 ?atime) crlf)
            )
            (assert (implementation ?name ?nnum E =(+ ?score ?vote)))
    )
)

;Done voting - find implementation with maximum score
(defrule choose_implementation_max
        (declare (salience 7800))
        ?fact1 <- (implementation max ? ? ?maxval)      ;max so far
        ?fact2 <- (implementation ?name ?nnum E ?score) ;imp score
=>
        (retract ?fact2)
        (if (> ?score ?maxval)          ;new maximum
            then
                (retract ?fact1)        ;dump old maximum
                (assert (implementation max ?name ?nnum ?score))
        )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;everything is finally set, assert the final implementation choice and clean up
;
(defrule choose_implementation_final    ;assert the final imp choice
        (declare (salience 7100))
        ?fact1 <- (implementation max ?name ?nnum ?)    ;highest scoring imp
        ?fact2 <- (execute algorithm ?) ;dump old flag
=>
        (retract ?fact1 ?fact2)
        (assert (execute implementation ?name ?nnum))   ;start execution flag
        (fprintout t "Execute implementation " ?name " for node " ?nnum crlf
                "--------------------------------------------------------" crlf)
)

(defrule choose_implementation_clean    ;clean up unneeded facts
        (declare (salience 7050))
```

```
        ?fact1 <- (implementation $?)
=>
        (retract ?fact1)
)
```

disc.main3.rules

```
; DISC
;       Dynamic Intelligent Scheduling and Control
;               Main Loop Section 3
;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;       MAIN DISC LOOP
;       --------------
;
;       III)  Begin the execution of the selected implementation.
;               A) Remove selected implementation from the PEN list
;               B) Add implementation and partition to the ACE list
;               C) Mark the partition as active
;               D) Tell the OS to start execution
;
;               6000 <= salience <= 6999
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defrule total_overhead_time            ;set up calculation for total overhead time
        (declare (salience 6960))
        (execute implementation $?)     ;an implementation has been chosen
=>
        (assert (TOTAL_LFR_TIME 0))     ;total load/format/reallocate time
)

(defrule load_implementation_data       ;load, reformat, and/or reallocate the data
        (declare (salience 6950))
        (execute implementation ?name ?)        ;implementation to be run
        ?fact1 <- (Do ?what ?name ?time $?rest) ;do necessary operation
        ?fact2 <- (TOTAL_LFR_TIME ?ttime)       ;total overhead so far
=>
        (retract ?fact1 ?fact2)
        (assert (TOTAL_LFR_TIME =(+ ?ttime ?time)))
        (fprintout t ?what " is running for " ?name crlf
                "    with time " ?time crlf "    and data " $?rest crlf)
)

(defrule implementation_data_clean      ;clean up unused implementation data
        (declare (salience 6940))
        ?fact1 <- (Do $?)                       ;no longer needed
=>
        (retract ?fact1)
)

(defrule start_execution        ;clean up things and start the implementation
        (declare (salience 6900))
        ?fact1 <- (execute implementation ?imp ?nnum)
        ?fact2 <- (PEN ?nnum)                   ;no longer a PEN algorithm
        ?fact6 <- (NUM_PEN ?numpen)             ;number of PEN algs
        ?fact5 <- (schedule next ?partnum)      ;partition number
        ?fact3 <- (Partition ?partnum $?)       ;partition to use
        ?fact4 <- (PARTITION I ?size ?partnum)  ;partition state
        ?fact7 <- (NUM_IP ?numip)               ;number of idle partitions
        (DB ?imp mode ?imode)                   ;partition mode
        (NODE ?nnum params ? $?pdata)           ;partition data
        (DB ?imp out_alloc $?palloc)            ;partition output allocation
        (DB ?imp out_format $?pformat)          ;partition output format
```

disc.main3.rules

```
        (DB ?imp etime ?size ?exectime)          ;expected execution time
        ?fact8 <- (TIMER ?timer)                 ;scheduling start time
        (SYS_TIME ?systime)                      ;internal time
        ?fact9 <- (TOTAL_LFR_TIME ?otime)        ;overhead time
        ?fact10 <- (DONE_TIME $?runlist)         ;what is running
=>
        (retract ?fact1 ?fact2 ?fact3 ?fact4 ?fact5)
        (retract ?fact6 ?fact7 ?fact8 ?fact9 ?fact10)
        (if (= ?exectime 1)
            then                                 ;unknown time - choose random time
                (bind ?exetime (+ (/ 8192 ?size) (random (- 2048 ?size))))
            else
                (bind ?exetime ?exectime)        ;known time
        )
        (assert (ACE ?nnum ?partnum))            ;Algorithm Currently Executing list
        (assert (NUM_PEN =(- ?numpen 1)))        ;1 less PEN algorithm
        (assert (NUM_IP =(- ?numip 1)))          ;1 less idle partition
        (if (neq (length $?palloc) 1)            ;more than 1 output or no outputs
            then                                 ;put in dummy data
                (bind ?data NONE)
                (bind ?alloc none)
                (bind ?format none)
            else                                 ;put in real data
                (bind ?alloc (nth 1 $?palloc))
                (bind ?format (nth 1 $?pformat))
                (if (eq (member M $?pdata) 0)
                    then                         ;labeled as output
                        (bind ?data (nth (- (member O $?pdata) 1) $?pdata))
                    else                         ;labeled as modified input
                        (bind ?data (nth (- (member M $?pdata) 1) $?pdata))
                )
        )
        (assert (Partition ?partnum ?imode ?data ?alloc ?format))
        (assert (PARTITION A ?size ?partnum))    ;partition now active
        (assert (DONE_TIME $?runlist ?nnum =(+ ?otime ?exetime ?systime)))
        (fprintout t "Implementation " ?imp " of node " ?nnum " is running" crlf
                   "    in partition "  ?partnum " of size " ?size crlf
                   "    with data " $?pdata crlf
                   "    and expected execution time of " ?exetime crlf)
        (format t "Algorithm scheduling time: %d%n" (- (numfired) ?timer))
)
```

disc.main4.rules

```
; DISC
;       Dynamic Intelligent Scheduling and Control
;               Main Loop Section 4
;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;       MAIN DISC LOOP
;       --------------
;
;       IV)  An implementation has finished, set up to restart DISC loop
;               A) Remove implementation from the ACE list,
;                  add it to the FEX list iff it is not a C node that
;                  jumps backwards (continues).
;               B) Mark partition as idle
;               C) Update the PEN list (check each node to see if all
;                  parents are done)
;                       1) If "A" node, add children
;                       2) If "C" node, add TRUE/FALSE children depending
;                          on the results of the condition
;               D) Signal that scheduling can begin again
;
;               5000 <= salience <= 5999
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defrule check_for_finish                 ;see if any algorithms are done now
        (declare (salience 5900))
        (ACE $?)                          ;something is running
        ?fact1 <- (SYS_TIME ?systime)     ;current system time
        ?fact2 <- (DONE_TIME $?runlist) ;list of running algs
=>
        (retract ?fact1)
        (bind ?which (finished ?systime $?runlist))     ;look for finished algs
        (if (= ?which 0)
            then                          ;nothing is done
                (assert (HOLD_TIME =(+ 10 ?systime)))   ;try again later
            else                          ;something finished
                (assert (HOLD_TIME ?systime))   ;suspend the timer
                (bind ?done (nth ?which $?runlist))
                (assert (DONE ?done))   ;show algorithm as done
                (retract ?fact2)        ;new running list
                (bind $?newlist (mv-delete ?which $?runlist))   ;remove alg num
                (assert (DONE_TIME =(mv-delete ?which $?newlist)))  ;remove alg time
                (if (>= (random 2) 1)   ;random results
                    then
                        (assert (RESULTS TRUE))
                    else
                        (assert (RESULTS FALSE))
                )
                (fprintout t "Algorithm " ?done " is now finished." crlf)
        )
)

(defrule schedule_the_next                ;keep scheduling
        (declare (salience 5100))
        ?fact1 <- (HOLD_TIME ?htime)      ;system time
        (PEN ?)                           ;something to schedule
=>
```

```
        (retract ?fact1)
        (assert (SYS_TIME ?htime))      ;restart the timer
        (assert (schedule next))
        (assert (TIMER =(numfired)))    ;timing info
)

(defrule check_for_finish_again         ;nothing is done - bump system time
        (declare (salience 5050))
        ?fact1 <- (HOLD_TIME ?htime)    ;current system time
=>
        (retract ?fact1)
        (assert (SYS_TIME ?htime))      ;restart the timer
)

(defrule finish_execution_a             ;add A node to FEX list if needed
        (declare (salience 5800))
        (DONE ?nnum)                    ;algorithm has finished
        (ACE ?nnum ?pnum)
        ?fact3 <- (PARTITION A ?size ?pnum)
        ?fact2 <- (FEX $?fexlist)       ;Finished EXecuting list
        (NODE ?nnum type A)             ;algorithm type node
        ?fact1 <- (NUM_IP ?numip)       ;number of idle partitions
=>
        (retract ?fact2 ?fact3 ?fact1)
        (assert (PARTITION I ?size ?pnum))      ;partition is now idle
        (assert (FEX $?fexlist ?nnum))  ;new FEX list
        (assert (NUM_IP =(+ ?numip 1)))
)

(defrule finish_execution_b             ;add C node to FEX list if needed
        (declare (salience 5800))
        (DONE ?nnum)                    ;algorithm has finished
        (ACE ?nnum ?pnum)
        ?fact3 <- (PARTITION A ?size ?pnum)
        (RESULTS ?results)              ;condition result
        ?fact2 <- (FEX $?fexlist)       ;Finished EXecuting list
        (NODE ?nnum type C)             ;condition type node
        (NODE ?nnum descend ? ? ?numtchild $?clist)     ;child list
        ?fact1 <- (NUM_IP ?numip)       ;number of idle partitions
=>
        (retract ?fact1 ?fact3)
        (assert (NUM_IP =(+ ?numip 1)))
        (assert (PARTITION I ?size ?pnum))      ;partition is now idle
        (if (|| (&& (eq TRUE ?results)
                    (> 0 ?numtchild))   ;condition true and TRUE branch not a loop
                (&& (eq FALSE ?results)
                    (> 0 (nth (+ 1 (abs ?numtchild)) $?clist))))
            then
                (retract ?fact2)
                (assert (FEX $?fexlist ?nnum))
        )
)

(defrule finish_execution_c             ;update the PEN list for A nodes
        (declare (salience 5700))
        (DONE ?dnum)                    ;algorithm that finished
        (NODE ?nnum parents $?plist)    ;parents of node to check
        (not (TAGGED ?nnum))            ;not already looked at
```

```
        (NODE ?dnum type A)                ;algorithm node
        (NODE ?dnum descend ? ? $? ?nnum $?)     ;node is in child list
        (FEX $?fexlist)
        ?fact1 <- (NUM_PEN ?numpen)        ;number of PEN algs
=>
        (assert (TAGGED ?nnum))
        (if (subset $?plist $?fexlist)  ;all parents are finished
            then
                (assert (PEN ?nnum))
                (retract ?fact1)
                (assert (NUM_PEN =(+ ?numpen 1)))
        )
)

(defrule finish_execution_d                ;update the PEN list for C nodes
        (declare (salience 5700))
        (DONE ?dnum)                       ;algorithm that finished
        (RESULTS ?results)                 ;condition results
        (NODE ?nnum parents $?plist)       ;parents of node to check
        (not (TAGGED ?nnum))               ;not already looked at
        (NODE ?dnum type C)                ;condition node
        (NODE ?dnum descend ? ? ?tnum $?clist)  ;child list
        (FEX $?fexlist)
        ?fact1 <- (NUM_PEN ?numpen)        ;number of PEN algs
=>
        (assert (TAGGED ?nnum))
        (if (|| (&& (eq FALSE ?results)
                    (member ?nnum (mv-subseq 1 (abs ?tnum) $?clist))
                    (subset $?plist $?fexlist))
                (&& (eq TRUE ?results)
                    (member ?nnum (mv-subseq (+ 2 (abs ?tnum))
                                             (length $?clist) $?clist))
                    (subset $?plist $?fexlist)))
            then
                (assert (PEN ?nnum))
                (retract ?fact1)
                (assert (NUM_PEN =(+ ?numpen 1)))
        )
)

(defrule finished_execution_clean_1        ;clean up
        (declare (salience 5250))
        ?fact1 <- (DONE ?nnum)
        ?fact2 <- (ACE ?nnum ?)
=>
        (retract ?fact1 ?fact2)
)

(defrule finished_execution_clean_2        ;get rid of unneeded facts
        (declare (salience 5200))
        ?fact1 <- (TAGGED ?)
=>
        (retract ?fact1)
)

(defrule finished_execution_clean_3        ;get rid of unneeded facts
        (declare (salience 5200))
        ?fact1 <- (RESULTS $?)
```

```
=>
        (retract ?fact1)
)
```

finished.c

```
/*
  finished

        This routine checks whether any partitions have become idle.
        If so, the list location is returned.  If not, 0 is returned.

        Input is the system time and the running process list.

        Must be called from CLIPS.
*/

#include <math.h>
#include "clips.h"

float finished()
{
      float systime;            /* internal system time */
      float which;              /* algorithm that is done */
      float time;               /* algorithm finish time */
      float thisnum, thistime;
      int numrun;               /* number of algorithms running */
      VALUE vptr;               /* variable info structure */
      int i;

      systime = rfloat(1);      /* get system time */

      runknown(2, &vptr);       /* get the alg-time list */
      numrun = get_vallength(vptr);     /* 2 * number of algorithms */

      time = HUGE;              /* dummy initial value */
      which = 0;
      for (i = 1; i <= numrun; i += 2) {
            thisnum = rmulfloat(&vptr, i);
            thistime = rmulfloat(&vptr, i + 1);
            if (thistime > systime) continue; /* alg not done yet */
            if (thistime < time) { /* lowest done alg so far */
                  time = thistime;
                  which = i;
            }
      }

      return which;
}
```

initialize.c

```
/*
  initialize - set up the CLIPS fact list by reading in, reducing, and
        counting the descendants of the data dependency graph.  The nodes
        in the RDDG are then asserted as facts

  input:
        none - will read file name from CLIPS

  output:
        1 - initialization successful
        0 - initialization error

  side effects:
        For each node, output is asserted to CLIPS as facts of the form:
        NODE <n> name <name>
        NODE <n> type <type>
        NODE <n> priority <priority>
        NODE <n> params <#_params param_1 param_type_1 ... param_n param_type_n>
        if node is "condition" type:
                NODE <n> descend num_descend rel_op num_T_nodes T_node_# ...
                                num_F_nodes F_node_# ...
        if node is "algorithm" type
                NODE <n> descend num_descendants num_children node_# ...
*/

#include <stdio.h>
#include "clips.h"              /* CLIPS header for interface */

#define TEMP_FILE "##RDDG.TEMP##" /* temp file name for RDDG */

float initialize ()
{
      int retval;               /* hold the return values */
      int rddg();               /* reduce the data dependency graph */
      int assert_RDDG();        /* assert the RDDG nodes */

      if (num_args() != 1) return 0; /* invalid init call */

      retval = rddg(rstring(1),TEMP_FILE); /* reduce the DDG */

      if (retval) retval = assert_RDDG(TEMP_FILE); /* stuff facts in CLIPS */

      unlink(TEMP_FILE);        /* remove temp file */
      return retval;
}
```

letter.c

```
/*
   letter
        This routine accepts a string argument and
        returns the first letter of the string.

        Must be called from CLIPS.
*/

#include "clips.h"

char letter()
{
     return *rstring(1);                /* return the first letter */
}
```

numfired.c

```
/*
  numfired()

  returns the number of rules fired so far
*/

int num_rules_fired;            /* incremented in engine.c of CLIPS */

float numfired() {
      return (float)num_rules_fired;
}
```

p_compact.c

```
/*
   p_compact

        Compacts the system to create idle partitions for each
        of the PEN algorithms.

        Input parameter is the number of PEN algorithms.
        Also, the file _SYS_INFO_ contains the current system state.

        Asserts into CLIPS facts of the form:
                (P_CMOVE old new)               -contents that have been moved
                (PARTITION status size num)     -new partitions
        so that the new system state is known to DISC

        Return value is the number of idle partitions now in the system
        (-1 if there is an error).

        Must be called from CLIPS.
*/

/*#define TEST                    define if non-CLIPS test wanted */

#include <stdio.h>
#ifndef TEST
#include "clips.h"
#endif

static int plist[32] = {        /* partition list */
      0, 16, 8, 24, 4, 20, 12, 28,
      2, 18, 10, 26, 6, 22, 14, 30,
      1, 17, 9, 25, 5, 21, 13, 29,
      3, 19, 11, 27, 7, 23, 15, 31
  };

struct {                        /* partition info structure */
      int num;                  /* partition number */
      int size;                 /* partition size */
      int inplace;              /* 1 if partition is in place */
} active[32], sactive[32];      /* max of 32 active partitions */
int numsize[5];                 /* number of partitions of a given size */
int numactive;                  /* number of active partitions */

#define STATEFILE "_SYS_INFO_"  /* system state information file */

int read_state();               /* read the system state file */
int clash();                    /* check for partition move clashes */

#ifndef TEST
float p_compact()
#else
float p_compact(numpen)
int numpen;
#endif
{
      FILE *statfile;           /* system state file */
#ifndef TEST
      int numpen;               /* number of PEN algorithms */
#endif
```

```
      int i, j, k, index;
      int lowest, low;          /* lowest available partition */
      unsigned short frees;     /* used to merge partitions */
      struct FNODE {            /* linked list of free partitions */
            int num;            /* partition number */
            struct FNODE *next;
      } *this;
      struct FHEAD {            /* header node for free partition lists */
            int num;            /* number of partitions of a given size */
            int size;           /* size of partitions */
            struct FNODE *list; /* list of free partitions */
      } fhead[5];
#define FNULL ((struct FNODE *)0) /* null node pointer */
      char asbuf[128];          /* asserted fact buffer */

#ifndef TEST
      struct fact *assert();    /* CLIPS fact assertion routine */

      numpen = (int)rfloat(1);  /* get the number of free partitions desired */
#else
#define assert(s) puts(s)
#endif
      /* read the system state info */
      if ((statfile = fopen(STATEFILE, "r")) == NULL) return -1.0;
      if (read_state(statfile)) return -1.0;

      /* sort the partitions by size - largest first */
      index = 0;
      k = 4;
      for (i = 512; i >=32; i /= 2) {
            numsize[k] = 0;
            for (j = 0; j <= numactive; j++) {
                  if (active[j].size != i) continue;
                  sactive[index].size = active[j].size;
                  sactive[index++].num = active[j].num;
                  numsize[k]++;
            }
            k--;
      }

      /* mark the partitions that are in place */
      lowest = 0;
      index = 0;
      k = 4;
      for (i = 512; i >=32; i /= 2) { /* compact - largest first */
            for (j = 0; j < numsize[k]; j++) {
                  if ((plist[sactive[index + j].num] >= lowest) &&
                        (plist[sactive[index + j].num] <=
                         (lowest + (numsize[k]-1) * i/32)))
                    sactive[j + index].inplace = 1;
                  else
                    sactive[j + index].inplace = 0;
            }
            lowest += numsize[k] * i / 32;
            index += numsize[k--];
      }

      /* compact the system going from largest to smallest active partitions */
```

```
lowest = 0;
index = 0;
k = 4;
for (i = 512; i >=32; i /= 2) { /* compact - largest first */
      low = lowest;
      for (j = 0; j < numsize[k]; j++) {
            if (sactive[index + j].inplace) {
                  sprintf(asbuf, "PARTITION A %d %d", i,sactive[index+j].num);
                  assert(asbuf);
            }
            else {
                  while (clash(plist[low], index, numsize[k]))
                    low += i / 32;
                  sprintf(asbuf, "PARTITION A %d %d", i, plist[low]);
                  assert(asbuf);
                  sprintf(asbuf,"P_CMOVE %d %d",sactive[index+j].num,
                          plist[low]);
                  assert(asbuf);
                  sactive[index + j].num = plist[low];
                  low += i / 32;
            }
      }
      lowest += numsize[k] * i / 32;
      index += numsize[k--];
}

/* make free partitions according to the number of partitions desired */
/* is #_PEN >= #_MC_free, then use each MC separately */
if (numpen >= (32 - lowest)) {
      index = 32 - lowest;
      for (i = lowest; i < 32; i++) {
            sprintf(asbuf, "PARTITION I 32 %d", plist[i]);
            assert(asbuf);
      }
      return index;        /* all possible partitions created */
}

/* make free partitions as large as possible */
index = 32;
j = 0;                     /* count number of free partitions */
k = 512;
frees = 32 - lowest;       /* number of free MCs */
for (i = 4; i >= 0; i--) { /* initialize the free lists */
      fhead[i].size = k;
      if ((frees >> i) & 1) { /* size exists - store it*/
            j++;
            fhead[i].num = 1;
            fhead[i].list = (struct FNODE *)malloc(sizeof(struct FNODE));
            fhead[i].list->next = FNULL;
            fhead[i].list->num = plist[index - k / 32];
            index -= k / 32;
      }
      else {               /* size doesn't exist - mark as blank */
            fhead[i].num = 0;
            fhead[i].list = FNULL;
      }
      k /= 2;
}
```

```
        /* count number of free partitions */
        frees = j;

        /* if #_PEN <= #_free, then mark all partitions as they stand */
        if (numpen <= frees) {
              for (i = 0; i < 5; i++) {
                    if (fhead[i].num == 1) {
                          sprintf(asbuf, "PARTITION I %d %d", fhead[i].size,
                                  fhead[i].list->num);
                          assert(asbuf);
                    }
              }
              return frees;
        }

        while (1) {                 /* keep splitting partitions until done */
              /* if (#_PEN == #_free) or (all free partitions
                 are as small as possible), then use all partitions */
              if ((numpen == frees) || (fhead[0].num == frees)) {
                    for (i = 0; i < 5; i++) {
                          this = fhead[i].list;
                          while (this != FNULL) {
                                sprintf(asbuf, "PARTITION I %d %d", fhead[i].size,
                                        this->num);
                                assert(asbuf);
                                this = this->next;
                          }
                    }
                    return frees;
              }

              /* if #_PEN > #_free - split largest partition in half - try again */
              for (i = 4; !fhead[i].num; i--); /* find largest partition */
              this = fhead[i].list; /* delete node from this list */
              fhead[i].list = this->next;
              fhead[i].num--;

              this->next = fhead[--i].list; /* insert new nodes in list */
              fhead[i].list = this;
              fhead[i].num += 2;
              this = (struct FNODE *)malloc(sizeof(struct FNODE));
              this->next = fhead[i].list;
              fhead[i].list = this;
              this->num = plist[plist[this->next->num] + fhead[i].size / 32];
              frees++;
        }
}

/*****************************************************************************
  int read_state(FILE *infile)

  Read in the system state file pointed to by infile.

  Return 0 on success, 1 on some error.
*/
int read_state(infile)
FILE *infile;
```

```
{
        numactive = -1;

        while (!feof(infile)) {
                numactive++;
                fscanf(infile, "%d %d0, &(active[numactive].size),
                        &(active[numactive].num));
        }

        return 0;
}

/*****************************************************************************
  int clash(int low, int start, int count)

  return 1 if low is an inplace partition, 0 otherwise
*/
int clash(low, start, count)
int low, start, count;
{
        int i;

        for (i = 0; i < count; i++)
          if (sactive[start + i].num == low) return 1;
        return 0;
}

/*****************************************************************************/
#ifdef TEST
main(argc, argv)
int argc;
char **argv;
{
        printf("0umber of free partitions %d0, (int)p_compact(atoi(argv[1])));
}
#endif
```

p_split.c

```
/*
   p_split

        Splits an idle partition into a number of new partitions
        depending on the number of PEN algorithms there are.

        Input parameters are the partition number, the number of
        PEN algorithms, the partition size.

        Asserts into CLIPS facts of the form (PARTITION I size num)
        and (Partition num s none none none)
        so that the new partitions can be recognized by DISC.

        Return value is the number of partitions that have been created.

        Must be called from CLIPS.
*/

#include "clips.h"

static int plist[32] = {        /* partition list */
      0, 16, 8, 24, 4, 20, 12, 28,
      2, 18, 10, 26, 6, 22, 14, 30,
      1, 17, 9, 25, 5, 21, 13, 29,
      3, 19, 11, 27, 7, 23, 15, 31
  };

int floor2[33] = {              /* largest power of 2 less than or equal to n */
      0,                        /* 0 */
      1,                        /* 1 */
      2, 2,                     /* 2, 3 */
      4, 4, 4, 4,               /* 4, 5, 6, 7 */
      8, 8, 8, 8, 8, 8, 8, 8,   /* 8, 9, 10, 11, 12, 13, 14, 15 */
      16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, /*16<=n<=31 */
      32
  };

float p_split()
{
      int pnum;                 /* partition number */
      int numpen;               /* number of PEN algorithms */
      int psize;                /* partition size */
      int i, index, fl;
      int mcpp;                 /* number of MCs per partition */
      char asbuf[128];          /* asserted fact buffer */
      struct fact *assert();    /* CLIPS fact assertion routine */

      pnum = (int)rfloat(1);
      numpen = (int)rfloat(2);
      psize = (int)rfloat(3) / 32;

      index = plist[pnum];      /* index of partition pnum in plist */

      if (psize <= numpen) {    /* split into as many partitions as possible */
            for (i = 0; i < psize; i++) {
                  sprintf(asbuf, "PARTITION I 32 %d", plist[index + i]);
                  assert(asbuf);
                  sprintf(asbuf, "Partition %d s none none none", plist[index+i]);
```

```
            assert(asbuf);
            /* place system call here to make SIMD partition */
        }
        return psize;       /* 'psize' partitions have been created */
    }

    fl = floor2[numpen];        /* largest power of 2 <= numpen */
    mcpp = psize / fl;          /* number of MCs per partition */
    for (i = 0; i < (2 * fl - numpen); i++) {
        sprintf(asbuf, "PARTITION I %d %d", mcpp * 32, plist[index]);
        assert(asbuf);
        sprintf(asbuf, "Partition %d s none none none", plist[index]);
        assert(asbuf);
        /* place system call here to make SIMD partition */
        index += mcpp;      /* next partition start */
    }

    mcpp /= 2;                  /* the rest of the partitions are half the size */
    for (i = 0; i < (2 * (numpen - fl)); i++) {
        sprintf(asbuf, "PARTITION I %d %d", mcpp * 32, plist[index]);
        assert(asbuf);
        sprintf(asbuf, "Partition %d s none none none", plist[index]);
        assert(asbuf);
        /* place system call here to make SIMD partition */
        index += mcpp;      /* next partition start */
    }

    return numpen;              /* 'numpen' partitions have been created */
}
```

random.c

```
/*
  nrandom(int n)

  return a random number between 0 and n inclusive.
*/

#include "clips.h"

float nrandom()
{
      float n;

      n = rfloat(1);
      srandom((int)time(0));
      return (float)(n * (double)random() / 2147483647.0);
}
```

rddg.c

```
/*****************************************************************************
  rddg - reduce data dependency graph

  Input:
        infile - file name of graph to reduce
        outfile - file name for resulting graph

  Output:
        1 - conversion successful
        0 - conversion error

  Side Effects:
        Reduced graph is sent to the file passed as the second
        parameter in the same format as the input except that the
        number of descendants is added as the last field and
        nodes in a loop have a negative number of descendants.

  Input File Format:
        n               {'n' is a unique integer >= 0}
        node_type priority      {"A" for an algorithm, "C" for a condition,
                                 0 - 9 for condition (9 is max)}
        name            {as listed in the DB}
        parameter list  {# of params, p1, p1type, ..., pn, pntype
                         p#type is "C", "I", "M", "O" for condition, input,
                         mod input, output}
        if the node is a condition node
                then condition, number of TRUE nodes and list of TRUE nodes
                        number of FALSE nodes and list of FALSE nodes
                else number of children and list of child nodes
*/

#include <stdio.h>
#include <math.h>

#define MAXNAME 30              /* max function name length */
#define MAXPARAMS 30            /* max number of parameters to function */
#define MAXCHILDREN 30          /* max number of children */
#define MAXNODES 100            /* max number of nodes in DDG */

struct NODE {                   /* node of the unreduced DDG */
      char type;                /* 'A' or 'C' */
      int priority;             /* 0 - 9 priority value (9 is max) */
      char name[MAXNAME];       /* as listed in the DB */
      int numparams;            /* number of parameters */
      char param[MAXPARAMS][MAXNAME]; /* parameters */
      char ptype[MAXPARAMS];    /* 'C', 'I', 'M', 'O' */
      char relop[3];            /* relational operator for functions */
      int numtrue;              /* number of TRUE nodes */
      int tgoto[MAXCHILDREN];   /* node numbers if condition true */
      int numfalse;             /* number of FALSE nodes */
      int fgoto[MAXCHILDREN];   /* node numbers if condition false */
      int numchild;             /* number of child nodes */
      int children[MAXCHILDREN]; /* children of node */
      int numdesc;              /* number of descendants (neg for loop) */
};

typedef struct NODE node;
node *nodelist[MAXNODES];       /* list of nodes */
```

rddg.c

```
/*************************************************************************/
int rddg (inname, outname)
char *inname, *outname;
{
      FILE *infile;              /* input file pointer */
      FILE *outfile;             /* output file pointer */
      int current;               /* current node location */
      int i;
      void areduce();            /* do the actual edge reduction for A nodes */
      void creduce();            /* do the actual edge reduction for C nodes */
      void newnode();            /* output a modified node */
      void markcycle();          /* mark all nodes in a cycle */

      /* open the input and output files */
      if ((infile = fopen(inname,"r")) == NULL) {
            perror(inname);      /* error exit */
            return 0;
      }

      if ((outfile = fopen(outname,"w")) == NULL) {
            fclose(infile);      /* close opened input file */
            perror(outname);     /* error exit */
            return 0;
      }

      /* set each node to NULL (non-existent) */
      for (i = 0; i < MAXNODES; i++)
        nodelist[i] = (node *) 0;

      /* read in each node */
      while (!feof(infile)) {
            /* read the node number and allocate the node */
            fscanf(infile,"%d0,&current);
            nodelist[current] = (node *)malloc(sizeof(node));

            /* init # of descendants and read the type, priority, and name */
            nodelist[current]->numdesc = 0;
            fscanf(infile,"%c%d235s",&(nodelist[current]->type),
                  &(nodelist[current]->priority), &(nodelist[current]->name[0]));

            /* read in the parameters */
            fscanf(infile,"%d",&(nodelist[current]->numparams));
            for (i = 0; i < nodelist[current]->numparams; i++)
                  fscanf(infile," %s %c",&(nodelist[current]->param[i][0]),
                        &(nodelist[current]->ptype[i]));

            /* the rest depends on the node type */
            if (nodelist[current]->type == 'C') {
                  /* conditional - read relop and goto nodes */
                  fscanf(infile,"%s %d",&(nodelist[current]->relop[0]),
                        &(nodelist[current]->numtrue));
                  for (i = 0; i < nodelist[current]->numtrue; i++)
                    fscanf(infile,"%d",&(nodelist[current]->tgoto[i]));
                  fscanf(infile,"%d", &(nodelist[current]->numfalse));
                  for (i = 0; i < nodelist[current]->numfalse; i++)
                    fscanf(infile,"%d",&(nodelist[current]->fgoto[i]));
            }
```

```
            else {
                  /* algorithm - read children */
                  fscanf(infile,"%d",&(nodelist[current]->numchild));
                  for (i = 0; i < nodelist[current]->numchild; i++)
                    fscanf(infile,"%d",&(nodelist[current]->children[i]));
            }
      }

      /* reduce each node */
      for (i = MAXNODES - 1; i >= 0; i--) {
            if (nodelist[i] != (node *) 0) {
                  if (nodelist[i]->type == 'A') /* remove unwanted edges */
                    areduce(nodelist[i], i); /* algorithm type */
                  else
                    creduce(nodelist[i], i); /* condition type */
            }
      }

      /* mark the cycle */
      for (i = MAXNODES - 1; i >= 0; i--) {
            if (nodelist[i] != (node *) 0)
              if (nodelist[i]->type == 'C')
                markcycle(nodelist[i], i);
      }

      /* output the new RDDG */
      for (i = MAXNODES - 1; i >= 0; i--)
        if (nodelist[i] != (node *) 0)
          newnode(nodelist[i], i, outfile);

      fclose(infile);           /* clean up and exit */
      fclose(outfile);
      return 1;
}

/*****************************************************************************
  areduce
        reduce one algorithm node of the DDG

        INPUT: pnode - pointer to the node to reduce
               num   - node number

        SIDE EFFECT: the node is reduced in place
*/

void areduce(pnode, num)
node *pnode;
int num;
{
      int descend[MAXNODES];    /* current set of descendants */
      int ndesc, ndescn;        /* number of descendants in list */
      int i, j;
      int addnodes();           /* add nodes to a list if not already there */
      int contained();          /* look for duplicate elements */

      /* start out by setting old D-list to the children of the node */
      ndesc = 0;
      for (i = 0; i < pnode->numchild; i++)
```

```
        if (pnode->children[i] > num) /* don't add cycle children */
          ndesc += addnodes(pnode->children[i], descend, ndesc);


      /* main loop to successively build descendant list */
      do {
            ndescn = 0; /* init to no new children */
            for (i = 0; i < ndesc; i++)
              ndescn += addnodes(descend[i], descend, ndesc + ndescn);
            ndesc += ndescn;

      } while (ndescn > 0);

      for (i = 0; i < pnode->numchild; i++)
        if (contained(pnode->children[i],descend,ndesc))
          pnode->children[i] = -1;

      pnode->numdesc = ndesc;
}

/*************************************************************************
  creduce
        reduce one condition node of the DDG

        INPUT: pnode - pointer to the node to reduce
               num   - node number

        SIDE EFFECT: the node is reduced in place
*/

void creduce(pnode, num)
node *pnode;
int num;
{
      int descend[MAXNODES];     /* current set of descendants */
      int ndesc, ndescn;         /* number of descendants in list */
      int i, j;
      int addnodes();            /* add nodes to a list if not there */
      int contained();           /* look for duplicate elements */

      /* start out by setting old D-list to the TRUE children of the node */
      ndesc = 0;
      for (i = 0; i < pnode->numtrue; i++)
        if (pnode->tgoto[i] > num) /* only add non-cycle children */
          ndesc += addnodes(pnode->tgoto[i], descend, ndesc);

      /* main loop to successively build descendant list */
      do {
            ndescn = 0; /* init to no new children */
            for (i = 0; i < ndesc; i++)
              ndescn += addnodes(descend[i], descend, ndesc + ndescn);
            ndesc += ndescn;

      } while (ndescn > 0);

      for (i = 0; i < pnode->numtrue; i++)
        if (contained(pnode->tgoto[i],descend,ndesc))
          pnode->tgoto[i] = -1;
```

```
      pnode->numdesc = ndesc;

      /* now set old D-list to the FALSE children of the node */
      ndesc = 0;
      for (i = 0; i < pnode->numfalse; i++)
        if (pnode->fgoto[i] > num) /* only add non-cycle children */
          ndesc += addnodes(pnode->fgoto[i], descend, ndesc);

      /* main loop to successively build descendant list */
      do {
            ndescn = 0; /* init to no new children */
            for (i = 0; i < ndesc; i++)
              ndescn += addnodes(descend[i], descend, ndesc + ndescn);
            ndesc += ndescn;

      } while (ndescn > 0);

      for (i = 0; i < pnode->numfalse; i++)
        if (contained(pnode->fgoto[i],descend,ndesc))
          pnode->fgoto[i] = -1;
      pnode->numdesc += ndesc;
}

/*************************************************************************
  addnodes
        add new child nodes to the list only if they don't already exist

        INPUT: nodenum - node whose children should be added
               list - list of nodes to add children to
               n - number of elements already in list

        OUTPUT: list - list with children added

        RETURN: number of children added to the list
*/
int addnodes(nodenum, list, n)
int nodenum;
int list[];
int n;
{
      int i;
      int count = 0;            /* number of nodes added to list */
      node * pnode;             /* temp node pointer */
      int contained();          /* list membership function */

      pnode = nodelist[nodenum]; /* add children of this node */
      if (pnode->type == 'A') {
            for (i = 0; i < pnode->numchild; i++)
                if ((pnode->children[i] != -1) &&
                    (!contained(pnode->children[i],list,n))) {
                      count++;  /* one more element */
                      list[n++] = pnode->children[i];
                }
      }
      else {
            for (i = 0; i < pnode->numtrue; i++)
              if ((pnode->tgoto[i] != -1) &&
                   (!contained(pnode->tgoto[i],list,n)) &&
```

```
                (pnode->tgoto[i] > nodenum)) {
                  count++;
                  list[n++] = pnode->tgoto[i];
              }
            for (i = 0; i < pnode->numfalse; i++)
              if ((pnode->fgoto[i] != -1) &&
                  (!contained(pnode->fgoto[i],list,n)) &&
                  (pnode->fgoto[i] > nodenum)) {
                  count++;
                  list[n++] = pnode->fgoto[i];
              }
      }

      return count;
}

/*****************************************************************************
  contained
        indicate whether or not an element is in an array

        INPUT: elem - element to look for
               list - list to search
               n - length of list

        RETURN: 1 - element contained in list
                0 - element not in list
*/

int contained(elem, list, n)
int elem;
int list[];
int n;
{
      int i;

      for (i = 0; i < n; i++)
        if ((elem != -1) && (elem == list[i])) break;

      if (i < n) return 1;

      return 0;
}

/*****************************************************************************
  newnode
        output a modified node to the specified file

        INPUT: pnode - pointer to node to output
               n - node number
               outfile - output file pointer
*/

void newnode(pnode, n, outfile)
node *pnode;
int n;
FILE *outfile;
{
      int i, j, ctot;
```

```
        fprintf(outfile,"240d240c %d240s0, n, pnode->type, pnode->priority,
                pnode->name);
        fprintf(outfile,"%d",pnode->numparams);

        for (i = 0; i < pnode->numparams; i++)
          fprintf(outfile," %s %c ", pnode->param[i], pnode->ptype[i]);

        fprintf(outfile,"0);
        if (pnode->type == 'C') {
              fprintf(outfile,"%s", pnode->relop);

              j = 0;              /* find new TRUE child count */
              for (i = 0; i < abs(pnode->numtrue); i++)
                if (pnode->tgoto[i] != -1) j++;

              ctot = pnode->numdesc + j;
              if (pnode->numtrue < 0) {
                    j = -j; /* keep cycle flag */
                    ctot = -ctot;
              }
              fprintf(outfile," %d", j);
              for (i = 0; i < abs(pnode->numtrue); i++)
                if (pnode->tgoto[i] != -1)
                  fprintf(outfile," %d", pnode->tgoto[i]);

              j = 0;              /* find new FALSE child count */
              for (i = 0; i < abs(pnode->numfalse); i++)
                    if (pnode->fgoto[i] != -1) j++;

              if (ctot < 0) /* keep cycle flag */
                ctot -= abs(j);
              else
                ctot = (ctot + abs(j)) * ((j < 0) ? -1 : 1);

              fprintf(outfile," %d", j);
              for (i = 0; i < abs(pnode->numfalse); i++)
                if (pnode->fgoto[i] != -1)
                  fprintf(outfile," %d", pnode->fgoto[i]);

              fprintf(outfile,"240d",ctot);
        }
        else {
              j = 0;              /* find new child count */
              for (i = 0; i < abs(pnode->numchild); i++)
                if (pnode->children[i] != -1) j++;

              fprintf(outfile,"%d",j); /* output child list */
              for (i = 0; i < abs(pnode->numchild); i++)
                if (pnode->children[i] != -1)
                  fprintf(outfile," %d", pnode->children[i]);

              fprintf(outfile,"240d",
                      (pnode->numdesc + j) * ((pnode->numchild < 0) ? -1 : 1));
        }
}

/*****************************************************************
```

```
  markcycle

        Mark all nodes in a cycle with a negative number of descendants.

  Usage: (void) markcycle (pnode, num)

  Input: pnode - pointer to a condition node
         num   - input node number

  Output: none

  Side Effects: The nodes in a cycle are modified in place
*/

void markcycle (pnode, num)
node *pnode;
int num;
{
      int i;
      void mark();              /* recursive cycle marker */

      /* first do the TRUE children */
      for (i = 0; i < pnode->numtrue; i++)
        if (pnode->tgoto[i] < num) { /* cycle indicator */
              mark(pnode->tgoto[i], num);
              pnode->numtrue = -pnode->numtrue;
        }

      /* now do the FALSE children */
      for (i = 0; i < pnode->numfalse; i++)
        if (pnode->fgoto[i] < num) { /* cycle indicator */
              mark(pnode->fgoto[i], num);
              pnode->numfalse = -pnode->numfalse;
        }
}

/*********************************************************************

  mark

        Recursively step through a cycle and mark each node

  Usage: (void) mark (nodenum, goal)

  Input: nodenum - number of node to mark
         goal    - place to stop marking

  Output: none

  Side Effects: The nodes in a cycle are modified in place
*/

void mark (nodenum, goal)
int nodenum, goal;
{
      int i;
      node *pnode;              /* pointer to node to process */
```

```
        void mark();                /* for recursive call */

        if (nodenum == goal) return; /* done with cycle for this path */
        pnode = nodelist[nodenum]; /* pointer to node to mark */

        if (pnode->type == 'A') { /* algorithm node */
              if (pnode->numchild > 0) /* only mark if not already marked */
                pnode->numchild = -pnode->numchild;
              for (i = 0; i < abs(pnode->numchild); i++) { /* mark children */
                    if (pnode->children[i] == -1) continue;
                    if (pnode->children[i] > nodenum) /* don't redo a cycle */
                      mark(pnode->children[i], goal);
              }
        }
        else {                      /* condition node */
              /* do TRUE nodes first */
              if (pnode->numtrue > 0) /* only mark if not already marked */
                pnode->numtrue = -pnode->numtrue;
              for (i = 0; i < abs(pnode->numtrue); i++) { /* mark all children */
                    if (pnode->tgoto[i] == -1) continue;
                    if (pnode->tgoto[i] > nodenum) /* don't redo a cycle */
                      mark(pnode->tgoto[i], goal);
              }

              /* do FALSE nodes next */
              if (pnode->numfalse > 0) /* only mark if not already marked */
                pnode->numfalse = -pnode->numfalse;
              for (i = 0; i < abs(pnode->numfalse); i++) { /* mark children */
                    if (pnode->fgoto[i] == -1) continue;
                    if (pnode->fgoto[i] > nodenum) /* don't redo a cycle */
                      mark(pnode->fgoto[i], goal);
              }
        }
}
```

task.parameters

```
(Parameter pixels 1048576)
(Parameter pixels_per_row 1024)
(Parameter pixels_per_column 1024)
(Parameter bins 128)
(Parameter features 64)
(Parameter classes 16)
(Parameter number_of_points 25)
(Loop_weight -3)
(Load_constant 0.0025)
```

teval.c

```
/*
  teval
        This routine evaluates the expression given
        by substituting variable values and performing
        the math ops.

        The expr is in prefix notation with parentheses
        and variables are one char long.

        Must be called from CLIPS.
*/

#include <string.h>
#include <math.h>
#include "clips.h"

float teval()
{
      char *expr;               /* expression to evaluate */
      float vals[20];           /* values to substitute */
      char vars[21];            /* variables to substitute */
      int j, k, numvars;
      float eval();             /* does actual evaluation */
      VALUE vptr;               /* variable info structure */

      expr = rstring(1);        /* expression to evaluate */
      runknown(2, &vptr);       /* get the var-val list */
      numvars = get_vallength(vptr) / 2; /* number of subst vars */

      for (j = 0; j < numvars; j++)
        {
              vars[j] = *(rmulstring(&vptr,2 * j + 1)); /* put in list */
              vals[j] = rmulfloat(&vptr,2 * j + 2);    /* put in list */
        }
      return (eval(expr,vars,vals));    /* evaluate expr */
}

/*
  eval

        This routine evaluates the expression given
        by substituting variable values and performing
        the math ops.

        The expr is in prefix notation with parentheses
        and variables are one char long.
*/

float eval(expr,vars,vals)
char *expr;                     /* expression to evaluate */
char vars[];                    /* list of variable names */
float vals[];                   /* list of variable values */
{
      int i;
      char *sexpr1, *sexpr2;    /* subexpression pointers */
      int plevel;               /* paren level */
      float res;                /* temp results */
      float eval();             /* for recursive call */
```

```
    switch (*expr)              /* process the expr */
      {
          case '0':             /* just return number */
          case '1':
          case '2':
          case '3':
          case '4':
          case '5':
          case '6':
          case '7':
          case '8':
          case '9':
            sscanf(expr,"%f",&res);
            return(res);
          case '(':                               /* eval expr */
            sexpr1 = &expr[2];          /* find 1st sexpr */
            while (*sexpr1 == ' ') sexpr1++;
            sexpr2 = &sexpr1[1];
            if (*sexpr1 != '(')      /* find sexpr2 */
              {
                    while (*sexpr2 != ' ') sexpr2++;
                    while (*sexpr2 == ' ') sexpr2++;
              }
            else
              {
                    plevel = 1;         /* bypass parens */
                    while (plevel)
                      {
                            if (*sexpr2 == '(') plevel++;
                            else if (*sexpr2 == ')') plevel--;
                            sexpr2++;
                      }
                    while (*sexpr2 == ' ') sexpr2++;
              }

            switch (expr[1])  /* process by op type */
              {
                  case '+':
                    return (eval(sexpr1,vars,vals) + eval(sexpr2,vars,vals));
                  case '-':
                    return (eval(sexpr1,vars,vals) - eval(sexpr2,vars,vals));
                  case '*':
                    return (eval(sexpr1,vars,vals) * eval(sexpr2,vars,vals));
                  case '/':
                    return (eval(sexpr1,vars,vals) / eval(sexpr2,vars,vals));
                  case 'l':   /* (l a b) means log base a of b */
                    return ((float) (log((double) eval(sexpr2,vars,vals)) /
                                     log((double) eval(sexpr1,vars,vals))));
              }

          default:              /* must be a variable */
            i = 0;
            while (vars[i] != *expr) i++;      /* find its index */
            return(vals[i]);
      }
}
```

tics.c

```
/*
  tics()

  returns the process cpu time in microseconds as a float
*/

#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>

float tics() {
      struct rusage usage;
      unsigned long t2;

      getrusage(0, &usage);
      t2 = usage.ru_stime.tv_sec -
           ((long)((long)(usage.ru_stime.tv_sec / 100)) * 100);
      t2 = 1000000 * t2 + usage.ru_stime.tv_usec;

      return (float) t2;
}
```

usrfuncs.c

```
/*************************************************************/
/* USRFUNCS:  The function which informs CLIPS of any user   */
/*   defined functions.  In the default case, there are no   */
/*   user defined functions.  To define functions, either    */
/*   this function must be replaced by a function with the   */
/*   same name within this file, or this function can be     */
/*   deleted from this file and included in another file.    */
/*   User defined functions may be included in this file or  */
/*   other files.                                            */
/*   Example of redefined usrfuncs:                          */
/*     usrfuncs()                                            */
/*       {                                                   */
/*        define_function("fun1",'i',fun1,"fun1");           */
/*        define_function("other",'f',other,"other");        */
/*       }                                                   */
/*************************************************************/

#include "clips.h"

usrfuncs()
  {
        extern float teval();   /* evaluate a math expression */
        extern float initialize(); /* init the data dependancy graph */
        extern float finished(); /* check for finished algorithms */
        extern char letter();   /* return the first letter of a string */
        extern float p_split(); /* split up a partition */
        extern float p_compact(); /* compact the state of the machine */
        extern float tics();    /* return the time */
        extern float nrandom(); /* return random numbers */
        extern float numfired(); /* return the number of rules fired so far */

        define_function("teval",'f',teval,"teval");
        define_function("initialize",'f',initialize,"initialize");
        define_function("finished",'f',finished,"finished");
        define_function("letter",'c',letter,"letter");
        define_function("p_split",'f',p_split,"p_split");
        define_function("p_compact",'f',p_compact,"p_compact");
        define_function("tics",'f',tics,"tics");
        define_function("random",'f',nrandom,"nrandom");
        define_function("numfired",'f',numfired,"numfired");
  }
```